AgentOps: Evals, Observability & Cost by Michaël Bettan

Definition: AgentOps (AI Evaluation, Observability, Runtime Engineering, and Production FinOps) is the operational control plane that transforms a probabilistic language model and multi-step tool loop into a reliable, auditable, and economically viable production service. Because generative agents sample tokens stochastically, branch dynamically across external APIs, and mutate state over multi-turn trajectories, classical software verification (binary unit tests and HTTP `200 OK` uptime checks) cannot detect silent quality regressions, "right answer, wrong trajectory" policy violations, or heavy-tail cost explosions.

Why production AI systems fail without an AgentOps control plane:

- **Silent quality & trajectory regressions:** A prompt edit, retrieval tweak, or unpinned model alias can degrade factual faithfulness by 20%—or skip a mandatory identity-verification tool—while every HTTP call still returns `200 OK`.
- **Compounding multi-step unreliability:** A single-step accuracy of 90% collapses to **59.0%** across a 5-step sequential agent workflow (0.90^5).
- **Integration seam failures:** Individual prompts and tools pass isolated tests at >95%, yet the end-to-end pipeline fails because Component A's output format, truncation limit, or error handling violates Component B's input assumptions.
- **Runtime & state mismatches:** Deploying a stateful orchestrator to an ephemeral scale-to-zero container without externalized session storage wipes in-flight memory, while attempting to host standalone MCP servers or custom GPUs on opinionated agent runtimes hits hard platform boundaries.
- **Unbounded heavy-tail economics:** Sizing budgets on median (`p50`) token usage leads to 4×–15× cost overruns when multi-agent fan-out, `thinking_level=HIGH` reasoning loops, tool retries, and mandatory human-review labor hit production scale.

> **MENTAL MODEL**
>
> Prompts and model weights are **probabilistic components**; your evaluation harness, simulation sandbox, OpenTelemetry trace pipeline, runtime boundary, resilience stack, and FinOps ledger form the **deterministic containment vessel** around them. You cannot eliminate variance inside the model, so you bound, measure, optimize, and price it at every boundary.

The 5-Pillar AgentOps Operational Loop:

1. **Specify, Simulate & Grade (Offline Evals):** Curate a stratified golden dataset, climb the 3-tier grading ladder (Deterministic Code → Calibrated Autoraters / Adaptive Rubrics → Human Experts), and stress-test multi-turn paths with **2-Step Agent Simulation** and **Environment Simulation**.
2. **Verify Seams & Consistency (`pass^k` & Test Pyramid):** Gate releases on worst-case consistency (`pass^k`), trajectory invariants (`trajectory_in_order_match`), and coupled Level 3 integration seam tests.
3. **Observe, Cluster & Optimize (Telemetry & Flywheel):** Instrument PII-safe **OpenTelemetry** `gen_ai.*` spans and dual-attribution audit logs (`is_agentic=true`), monitor live traffic via **Online Evaluation Monitors**, group regressions with **Failure Cluster Analysis**, and close the loop with **Agent Optimizer / Prompt Optimizer**.
4. **Select Runtimes & Isolate Faults (Scale & Resilience):** Match workloads to **Agent Runtime**, **Cloud Run**, or **GKE** (`Agent Substrate` / `Agent Sandbox`), and wrap every model/tool hop in full-jitter backoff, idempotency keys, circuit breakers, and validated fallbacks.
5. **Roll Out Safely & Price the Tail (Shadow/A/B & FinOps):** Promote changes through read-only **Shadow Mode** and **4-Element A/B Tests**, model budgets on `p95` / `p99` heavy tails, and subtract the **Step 4 Human-Review Tax** to verify net enterprise ROI.

1. Evaluation Strategy & The 3-Tier Grading Hierarchy

Why "Vibe Checks" Fail in Production

Eyeballing 5 to 10 interactive examples ("vibe checking") is useful for initial prototyping, but it breaks down immediately as an engineering release gate:

- **Zero statistical power:** On a sample of 10 inputs, a 15% regression is indistinguishable from sampling noise.
- **Confirmation bias:** Developers naturally test happy-path phrasing and subconsciously avoid adversarial inputs, malformed documents, or ambiguous multi-intent queries.
- **Unseen trade-offs:** Tightening a prompt to stop false positives on one edge case frequently induces false refusals or format regressions on twenty others that nobody re-tested.

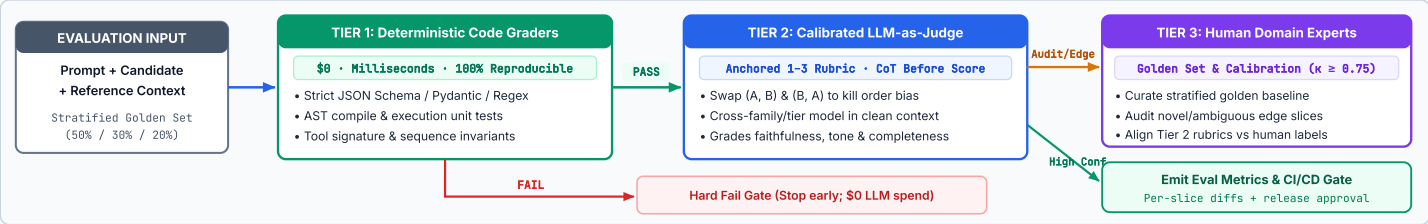
Eval-Driven Development (EDD) Lifecycle:

1. **Define "done" quantitatively first:** Specify target task completion rate, trajectory compliance, acceptable refusal rate, `p95` latency ceiling, and maximum cost per resolved task *before* iterating on prompts or switching models.
2. **Curate a stratified golden dataset:** Combine happy-path production queries (50%), boundary/edge cases such as empty retrieval, multi-intent prompts, and truncated inputs (30%), and adversarial/safety-probing inputs (20%).
3. **Version everything:** Pin the golden dataset version, autorater rubric version, system prompt template, tool schemas, Agent Registry snapshot, and exact model ID so every eval run is a single-variable experiment.
4. **Inspect per-case diffs, not just aggregate means:** Two prompts that both score 86% overall may fail on completely different slices; always inspect regressions by failure cluster and severity.

The 3-Tier Grading Ladder

Always climb the grading ladder from the cheapest, fastest, most deterministic check to the most expensive semantic review. Never pay an LLM judge or human annotator to verify a property that a 5-line function can check deterministically.

Tier	Grader Type	What It Evaluates	Speed & Cost	Reproducibility	Rule of Thumb
Tier 1	**Deterministic Code Graders**	Exact enum/label match, regex/forbidden strings, JSON Schema & Pydantic validation, AST/compile checks, SQL/code execution unit tests, tool-call sequence & argument bounds	Milliseconds · **$0**	**100% deterministic**	**Always run first.** Gate every CI/CD commit on Tier 1 properties.
Tier 2	**Calibrated Autoraters (LLM-as-Judge)**	Open-ended semantic quality: factual faithfulness to retrieved context, adaptive rubric satisfaction, tool-selection rationale, tone, policy compliance	Seconds · **Token cost per run**	High (when anchored + calibrated)	Use only for semantic qualities code cannot parse; calibrate against Tier 3.
Tier 3	**Human Domain Calibration**	Ambiguous domain edge cases, high-stakes medical/legal/financial accuracy, novel failure discovery, golden-set labeling, autorater calibration	Hours to days · **Highest ($50–$200+/hr)**	Subject to inter-rater variance	Use to build the golden set, audit slices, and measure Tier 2 agreement.



LLM-as-Judge Biases & Engineering Mitigations

Using an uncalibrated LLM as a grader introduces systematic measurement errors that can steer prompt engineering in the wrong direction.

Judge Bias	Mechanism & Symptom	Production Mitigation
Verbosity (Length) Bias	Judges consistently award higher scores to longer, polite, multi-paragraph responses even when they contain fluff, repetition, or subtle hallucinations.	Explicitly penalize unnecessary verbosity in the rubric; grade atomic factual claims (precision/recall) rather than holistic impression; normalize or cap length where appropriate.
Positional (Order) Bias	In pairwise A/B comparisons, judges systematically favor whichever response appears in slot A (primacy bias) or slot B (recency bias).	Run **every pairwise comparison twice** with swapped order `(A, B)` and `(B, A)`. Count a win only if the same candidate wins both passes; mark split decisions as a tie.
Self-Preference Bias	A model family or checkpoint favors outputs matching its own stylistic distribution and blind spots, overlooking reasoning errors it would make itself.	Grade with a **different model family or higher capability tier** than the generator, and always run the judge in a **fresh, isolated context window** containing only the rubric, source ground truth, and candidate output.
Score Inflation & Scale Drift	Asking an LLM to "rate quality from 1 to 100" or "1 to 10" yields arbitrary clusterings around 7–9 with low run-to-run consistency.	Use **binary (`Pass` / `Fail`) checks** decomposed per criterion (adaptive rubrics), or low-granularity **1–3 or 1–5 anchored scales** where every integer has an explicit behavioral definition.
Post-Hoc Rationalization	If the judge outputs a numeric score *first* and its explanation *second*, attention locks onto justifying the emitted token rather than evaluating the evidence.	Enforce structured output where the **`reasoning` / `critique` field precedes the `score` field** (Chain-of-Thought *before* judgment).

Calibrating the Autorater Against Human Ground Truth

Because a model-based autorater acts as a learned classifier, benchmark its reliability before trusting its scores:

1. **Build a human-labeled calibration set:** Have 2–3 domain experts independently grade 100–300 stratified examples using the exact same rubric. Measure **inter-annotator agreement** first (if humans disagree with each other, e.g., $\kappa < 0.40$, the rubric itself is ambiguous and must be rewritten before touching the LLM judge).
2. **Measure Autorater–Human Alignment:**
   - **Cohen's Kappa (κ):** Measures agreement on binary or categorical classifications while correcting for chance agreement ($\kappa = (p_o - p_e) / (1 - p_e)$). Target **$\kappa \geq 0.75$** before automating release gates.
   - **Spearman's Rank Correlation (ρ) / Kendall's τ:** Measures monotonic ranking alignment on ordinal 1–5 rubrics.
   - **Confusion Matrix (FPR / FNR):** Track false-pass rate (judge says Pass, human says Fail) separately from false-fail rate; in safety or compliance gates, tune the rubric to minimize false passes. | Rubric Evaluation Field | Validation Type | Autorater Target | | :--- | :--- | :--- | | `extracted_claims` | `list[str]` | Atomic factual claims made in the candidate response. | | `unsupported_claims` | `list[str]` | Claims from `extracted_claims` not directly supported by the source context. | | `critique` | `str` (CoT) | Step-by-step verification comparing each claim against the source text before scoring. | | `score` | `int` (1–3) | 1 = Any hallucinated/unsupported claim; 2 = Faithful but omits key context; 3 = Fully faithful and complete. |

COMMON TRAP

Never ask the same model context that generated an answer to critique and score its own output in the next conversational turn ("self-grading in-thread"). Because prior reasoning tokens and assumptions remain in `messages[]`, the model exhibits severe self-confirmation bias. Always invoke autoraters in a **clean, independent request**.

2. Managed Evaluation Frameworks, Autoraters & Simulation

In enterprise Google Cloud and multi-framework architectures, teams must choose the right evaluation tool for each stage of the developer inner loop, CI/CD pipeline, pre-release stress test, and live production monitor—rather than forcing a single tool to do everything.

Choosing the Right Evaluation Tooling Tier

Evaluation Tier & Tooling	Primary Mechanism	Speed & Variance	Best Fit & Architectural Boundary
1. Deterministic Harness Replay (`adk conformance test --mode=replay`)	Replays recorded LLM responses from `adk conformance record` baselines against live agent code, callbacks, and state transitions.	**< 5 seconds · $0 · Zero variance**	Every git commit / pre-commit hook verifying that callback guardrails, `state_delta` logic, and routing code did not regress.
2. In-Repo CI Gate (`adk eval`, `AgentEvaluator.evaluate`, `agents-cli eval run`)	ADK runs `*.test.json` files and `*.evalset.json` evalsets against live models, grading `test_config.json` criteria such as `tool_trajectory_avg_score` (`EXACT`, `IN_ORDER`, `ANY_ORDER`), rubric criteria, and custom metrics. Upgraded `agents-cli` scaffolds read `EvaluationDataset` files (`tests/eval/datasets/*-dataset.json`) with `tests/eval/eval_config.yaml` (migrated via `agents-cli scaffold upgrade` from earlier `tests/eval/evalsets/*.evalset.json` + `eval_config.json` layouts).	**1–5 minutes · Low cost**	Pull-request blocking gates inside GitHub Actions or Cloud Build for code-first ADK repositories (`tests/eval/`).
3. Cloud-Scale Managed Evaluation (**Agent Platform Gen AI Evaluation Service** — `client.evals`)	Framework-agnostic managed service (`EvalTask` / `client.evals.evaluate`) providing **Adaptive Rubrics**, **Custom Autoraters**, **2-Step Agent Simulation**, **Environment Simulation**, and **Online Evaluation Monitors**.	**Minutes to continuous · Managed scale**	Cross-language/cross-framework agents (ADK, LangGraph, Managed Agents, Cloud Run), model migrations (e.g., to `gemini-4-argon`), synthetic stress-testing, and live production trace scoring.

4-Way Evaluation Tool Matrix: `ADK evalset` vs. `agents-cli eval` vs. Batch `evaluate-offline` vs. Online Monitors `evaluate-online`

The tiers above say *when* to evaluate. This matrix maps *which execution surface* to use across local repositories, CI/CD pipelines, batch benchmarks, and live production traffic—including dataset-flag and reference-trajectory boundaries:

Dimension	**`ADK evalset`** (**`adk eval`** / **`AgentEvaluator`**)	**`agents-cli eval`** (**`eval run`** / **`compare`** / **`analyze`**)	Batch **`evaluate-offline`** (**`client.evals.evaluate`** / **`EvalTask`**)	Online Monitors **`evaluate-online`** (**`Online Evaluator`**)
Primary use case	Unit and trajectory regression tests inside an ADK repo; interactive session capture from `adk web`	Inner eval-fix loop, PR gates, release-candidate diffs (`eval compare`), and failure clustering (`eval analyze`) for `agents-cli` repos	Framework-agnostic cloud batch benchmarks, model migrations, **2-Step Agent Simulation**, **Environment Simulation**, and **Automatic Loss Analysis**	Scheduled (~10 min) continuous async scoring of sampled live production traces in Cloud Trace + Cloud Logging
Dataset & flag contract	`*.test.json` (single session; migrate legacy schemas via `AgentEvaluator.migrate_eval_data_to_new_schema`) and `*.evalset.json` + `test_config.json`	Upgraded **`EvaluationDataset`** JSON (`--dataset` / `tests/eval/datasets/*-dataset.json`) + `eval_config.yaml` *(earlier quickstart scaffolds used `tests/eval/evalsets/*.evalset.json` + `eval_config.json`)*	Tabular `EvaluationDataset` (SDK DataFrames, GCS JSONL/CSV, BigQuery, Agent Registry snapshots, or canonical OTel traces; **rejects raw ADK `*.evalset.json`**)	Live OpenTelemetry inference events (`EVENT_ONLY` capture); **no static ground-truth reference dataset**
Supported metric classes	`tool_trajectory_avg_score` (`EXACT`, `IN_ORDER`, `ANY_ORDER`), `response_match_score`, `final_response_match_v2`, `hallucinations_v1`, rubric & custom Python functions	Built-in Gen AI Evaluation metrics (`tool_use_quality`, `multi_turn_task_success`, `multi_turn_trajectory_quality`, `final_response_match`, `hallucination`, `safety`) + `custom_function` / `prompt_template`	**All families:** all 6 deterministic trajectory metrics, adaptive rubrics, static/pointwise/pairwise rubrics, `CustomMetric`, and `types.LLMMetric`	**Reference-free rubrics only** (`HALLUCINATION`, `TOOL_USE_QUALITY`, `FINAL_RESPONSE_QUALITY`, `SAFETY`, `MULTI_TURN_TASK_SUCCESS`); **cannot run reference trajectory metrics**
Where inference & grading run	Local dev machine or CI runner (`pytest` / `adk eval`); LLM judge criteria call Vertex AI	Agent runs locally/in CI; grading calls Gen AI Evaluation SDK (`execution: remote` sandboxes custom code server-side)	Managed Google Cloud evaluation backend (SDK or Console)	Managed project-level service agent (P4SA) polling Cloud Trace & Cloud Logging out of band
Primary cost driver	Agent tokens + judge tokens on CI slices	Agent tokens + judge tokens (`judge_model_sampling_count` 1–32)	Batch agent inference + judge LLM calls per rubric	**Zero agent token overhead**, but **5–31 judge LLM calls per sampled trace** (e.g., ~6 calls for `TOOL_USE_QUALITY`, **31 Flash calls** for `SAFETY` `safety_v3`; cap via sampling!)

TWO HIGH-FREQUENCY EVALUATION TRAPS

1. **Dataset-schema & CLI flag mismatch (`ADK evalset` vs. `agents-cli eval` vs. `evaluate-offline`):** Native `adk eval` consumes ADK `*.evalset.json` session files (`test_config.json`), whereas managed Batch `evaluate-offline` (`client.evals.evaluate()`) expects tabular `EvaluationDataset` rows (`predicted_trajectory` / `reference_trajectory` or canonical OTel traces) and rejects raw ADK `*.evalset.json` files. Likewise, upgraded `agents-cli eval` scaffolds read `EvaluationDataset` JSON under `tests/eval/datasets/*-dataset.json` (`eval_config.yaml`) and ignore legacy `tests/eval/evalsets/*.evalset.json` files until migrated via `agents-cli scaffold upgrade` (manually porting any seeded `session.state` , and using `Agent Evaluator.migrate_eval_data_to_new_schema` for older `*.test.json` files). 2. **Why `trajectory_exact_match` cannot run on live production traffic:** All 6 deterministic trajectory metrics compare executed tool calls against a labeled **ground-truth reference trajectory**. Live production conversations have no ground-truth reference trace per user query, so **Online Monitors (`evaluate-online`)** must score **reference-free autorater rubrics** (`HALLUCINATION` , `TOOL_USE_QUALITY` , `FINAL_RESPONSE_QUALITY` , `SAFETY` , `MULTI_TURN_TASK_SUCCESS`).

- **Anti-pattern:** writing a custom LLM judge for a property a built-in metric already scores (task success, hallucination, safety). Start with built-ins, and add `CustomMetric` or `types.LLMMetric` only for domain rules they miss.
- **Anti-pattern:** running untrusted custom metric code locally in CI with the runner's credentials. Set `execution: remote` in `eval_config.yaml` for sandboxed server-side execution or audit the code like any production dependency.

Gen AI Evaluation Service: Complete Metric Catalogue

The **Gen AI Evaluation Service** (`client.evals`) provides four distinct metric families spanning `$0` deterministic trajectory checks, adaptive per-prompt rubrics, static/pairwise rubrics, and custom code or LLM autoraters:

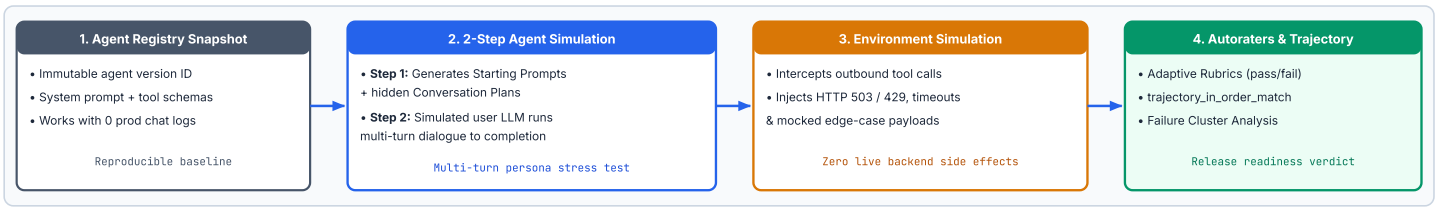
Metric Family	Reference Required?	Judge Cost Profile	Complete Canonical Metrics & Mechanics
1. Deterministic Trajectory Metrics *(6 built-ins)*	**Yes** (Golden tool trace; **offline/CI only**)	**`$0` judge LLM cost** (pure deterministic computation)	• `trajectory_exact_match` : exact tool names and exact sequential order (strict compliance gates). • `trajectory_in_order_match` : required milestone tools appear in order; extra read-only lookups allowed. • `trajectory_any_order_match` : all expected tools called in any order (validates parallel tool fan-out). • `trajectory_precision` : fraction of executed tool calls present in the reference trajectory. • `trajectory_recall` : fraction of reference tool calls actually executed by the agent. • `trajectory_single_tool_use` : binary check verifying one specific mandatory tool was invoked at least once.
2. Adaptive Rubric-Based Metrics *(Recommended semantic default)*	**No** (Reference-free; offline & online)	**Multi-call LLM judge** (e.g., `TOOL_USE_QUALITY` invokes **~5× Flash + 1× Pro** judge calls per trace)	Dynamically synthesizes a tailored list of prompt-specific pass/fail unit-test rubrics per input before grading: • `FINAL_RESPONSE_QUALITY` & `GENERAL_QUALITY` (end-to-end helpfulness & completeness). • `TOOL_USE_QUALITY` (tool selection, schema adherence, argument grounding). • `MULTI_TURN_TASK_SUCCESS` , `MULTI_TURN_TOOL_USE_QUALITY` , `MULTI_TURN_TRAJECTORY_QUALITY` (multi-turn dialogue & simulation grading). • `INSTRUCTION_FOLLOWING` (constraint & format obedience).
3. Static, Pointwise & Pairwise Rubrics	**No** (`HALLUCINATION` , `GROUNDING` , `SAFETY`) / **Yes** (`FINAL_RESPONSE_MATCH`)	**Multi-call LLM judge** (e.g., `SAFETY` default `safety_v3` invokes **31× Gemini 3.5 Flash** calls/trace where `1` = violation, `0` = clean; legacy `safety_v1` ran **10×** calls)	Fixed rubric criteria applied uniformly across every example: • `GROUNDING` (`0.0` – `1.0` + rationale): checks factual entailment against retrieved RAG chunks and tool outputs (*trap:* passes if RAG retrieves an outdated policy PDF and the agent cites it faithfully). • `HALLUCINATION` : labels sentences `supported` , `unsupported` , `contradictory` , or `disputed` relative to intermediate context (*trap:* omitting tool outputs from trace spans triggers false-positive flags). • `SAFETY` : policy, harm, and jailbreak compliance. • `FINAL_RESPONSE_MATCH` : semantic equivalence against a golden reference answer. • `PointwiseMetric` (single-output rubric score) & `PairwiseMetric` (head-to-head A/B win-rate with candidate order swapping to eliminate LLM judge position bias).
4. Custom Code & LLM Autoraters	**Configurable**	**Free** (`CustomMetric`) or **Judge tokens × sampling count** (`types.LLMMetric`)	• **`CustomMetric`** (**`CodeExecutionMetric`**): deterministic Python function (`EXACT_MATCH` , `ROUGE` , `BLEU` , regex, JSON Schema, SQL AST equivalence, forbidden-tool or token-budget checks; never waste `LLMMetric` quota on deterministic format checks). • **`types.LLMMetric`** : custom `prompt_template` , `result_parsing_function` , critique-before-score schema, and pinned `judge_model` calibrated against human domain experts (κ ≥ 0.75).

THE "RIGHT ANSWER, WRONG PATH" COMPLIANCE TRAP

Grading a customer-refund or healthcare agent solely with `FINAL_RESPONSE_QUALITY` or `MULTI_TURN_TASK_SUCCESS` will pass an agent that guessed or issued a refund *without* calling `verify_customer_identity` or `check_return_eligibility` first. Whenever policy mandates verification steps, **always pair outcome autoraters with `trajectory_in_order_match` , `trajectory_exact_match` , or `trajectory_single_tool_use`** (plus a forbidden-tool `CustomMetric`) in CI.

Pre-Launch Stress Testing: 2-Step Agent Simulation & Environment Simulation

Hand-written golden sets rarely exceed 50–200 single-turn prompts and miss multi-turn clarification loops or third-party API brownouts. Before launching—even when zero historical production logs exist—combine Agent Platform's two simulation engines:



1. **2-Step Agent Simulation:** Requires an immutable **Agent Registry** snapshot so every benchmark is reproducible. In **Step 1**, the simulator reads the agent's system instructions and tool declarations to synthesize realistic `Starting Prompts` paired with hidden `Conversation Plans` (e.g., an impatient traveler who only reveals their booking code when explicitly asked). In **Step 2**, a simulated user model conducts the multi-turn session and records full traces.
2. **Environment Simulation:** Wraps tool boundaries during evaluation to inject deterministic faults (`HTTP 503 Service Unavailable` , `429 RESOURCE_EXHAUSTED` , 12-second latency spikes, empty results) and mocked responses—proving your retry, fallback, and escalation guardrails work **without risking real side effects on production payment or ERP APIs**.

Safety evaluations, multi-turn red-teaming & guardrail shadow tuning

1. **Offline `SAFETY` rubric (`safety_v3` / `safety_v1` , `GA`):** Scores agent responses (`0.0` – `1.0` with rationale) across Hate Speech, Harassment, Dangerous Content, and Sexually Explicit harm policies before release.
2. **Multi-turn adversarial red-teaming (`UserSimulator` , `GA`):** Static prompt lists miss multi-turn social engineering. ADK's `UserSimulator` pairs a `UserPersona` (adversarial jailbreaker, confused caller, policy-bypassing insider) with `violation_rubrics` to probe the agent dynamically across turns, while `per_turn_user_simulator_quality_v1` verifies simulator fidelity.
3. **Model Armor false-positive shadow tuning (`INSPECT_ONLY` arrow `INSPECT_AND_BLOCK`):** Deploy **Model Armor** templates first in `INSPECT_ONLY` mode (`GA`), inspect `SanitizeOperationLogEntry` verdicts in Cloud Logging to measure false-positive hits on legitimate domain terminology, tune confidence thresholds (`MEDIUM_AND_ABOVE` vs. `HIGH`), and then promote to `INSPECT_AND_BLOCK` .

CI/CD evaluation gates: from pull request to canary

Each gate is cheaper and faster than the next, so most regressions die early and only release candidates pay for simulation and judge-heavy grading.

```
PR opened ──► Gate 1: trajectory + code metrics ──fail──► block merge
                 │ pass
merge ─────► Gate 2: simulation + rubric metrics on release candidate
                 │ pass
              Gate 3: compare RC vs. production ──regression──► analyze, fix
                 │ no regression
              Canary revision ──► online monitors ──► promote or shift back
```

1. **PR trajectory gate:** every pull request that touches prompts, tools or model settings runs `agents-cli eval run` (or `adk eval` / `pytest` with `AgentEvaluator` in an ADK-only repo) in CI, for example a pipeline created with `agents-cli infra cicd` . Cases replay recorded or mocked tool outputs, and the metrics are trajectory checks plus cheap code metrics, so the gate stays fast and deterministic enough to block merges.
2. **Release-candidate simulation and rubric gate:** after merge, deploy the candidate to staging and run simulated multi-turn scenarios (from `agents-cli eval dataset synthesize` or the Gen AI evaluation service user simulation) with environment simulation for tool faults. Grade with `multi_turn_task_success` , `multi_turn_trajectory_quality` , `hallucination` , `safety` and any calibrated custom autoraters, and enforce an absolute floor per metric.
3. **Regression compare against production:** grade the same frozen dataset against the current production revision and the candidate, then run `agents-cli eval compare --fail-on-regression` on the two result files. Fail the build when a critical slice drops beyond the agreed tolerance, even if the averages still clear their floors, and run `agents-cli eval analyze` to cluster what broke.
4. **Canary rollout:** ship the candidate as a new revision and route a small slice (e.g., 5%) via Agent Runtime `revisions update --traffic-split` or Cloud Run `--to-revisions` . **Two canary traps:** (a) keeping state in `InMemorySessionService` drops multi-turn history across split revisions, and (b) **Agent Runtime native revision `traffic-split` is bypassed when Agent Gateway controls routing** (split traffic at the gateway/backend layer instead). Online monitors and **Quality Alerts** (§05) watch the canary before full promotion.

```
# Gate 1 & Gate 3: offline golden gate + pairwise baseline diff + loss clustering
agents-cli eval run
agents-cli eval compare prod_results.json rc_results.json --fail-on-regression
agents-cli eval analyze --eval-result rc_results.json
# Canary rollout: shift 5% traffic to new Agent Runtime revision (direct routing)
gcloud beta ai reasoning-engines revisions update "$REV_V2" \
  --region="REGION" --project="PROJECT_ID" --traffic-split="REV<sub>V</sub>1=95,REV_V2=5"
```

Directional values: metric floors, regression tolerances and canary percentages are team choices, not Google defaults. The result file names above are placeholders.

- **Anti-pattern:** gating only on the average score. A two-point average gain can hide a collapse on one compliance slice; gate per slice and per critical metric.
- **Anti-pattern:** letting the judge model float between runs. Pin the judge model and sampling count so a score change reflects the agent, not the grader.
- **Anti-pattern:** running PR gates against live mutating tools. Use recorded fixtures or environment simulation so CI never sends real emails or refunds.

Golden set design checklist

A golden set is the frozen, versioned dataset every gate grades against. Cover each case type below and tag every case with its slice and origin so regressions can be traced.

Case type	What it catches	How to express it	Typical metrics
Trajectory cases	Skipped verification steps, wrong tool, bad arguments	Single prompt (`prompt` field in `agents-cli` , one turn in an ADK evalset) plus expected tool calls; assert required milestones in order, not full transcripts	`tool_trajectory_avg_score` (`IN_ORDER`), `tool_use_quality` , forbidden-tool code check
Continued-conversation cases	Mid-session regressions after earlier turns set context	`agent_data` turns ending with a user message (`agents-cli`) or a multi-turn evalset session (ADK)	`multi_turn_tool_use_quality` , `multi_turn_trajectory_quality`
Simulated multi-turn scenarios	Clarification loops, users who change their mind, goal abandonment	Starting prompt plus conversation plan and persona (`eval dataset synthesize --instruction ...` , ADK conversation scenario, service user simulation)	`multi_turn_task_success`
Abstention cases	Hallucinating when retrieval is empty, answering out-of-scope or unauthorized requests	Prompts with no supporting data or outside the agent's remit; expected outcome is a decline or a clarifying question	`hallucination` , custom refusal check
Adversarial cases	Prompt injection in tool output, jailbreaks, PII extraction	Injected strings in mocked tool results and user turns; expected outcome is refusal and no forbidden tool call	`safety` , forbidden-tool code check
Fault cases	Broken retry, fallback and escalation paths	Environment simulation returning `429` , `503` , timeouts, empty rows	Trajectory milestones on the fallback path
Loss-analysis backfill	Production failure classes the set never covered	Representative traces from Automatic Loss Analysis or `agents-cli eval analyze` clusters, scrubbed of PII and converted to cases	Whatever metric flagged the cluster

- **Held-out split:** keep a slice that neither engineers nor `agents-cli eval optimize` see during tuning, so prompt fixes generalize instead of memorizing the set.
- **Version with the code:** commit datasets and `eval_config.yaml` next to the agent so a revision and its gate always travel together.
- **Retire saturated cases:** when a slice scores near perfect for many releases, replace it with harder or fresher cases from the backfill loop.

3. Non-Deterministic Metrics: `pass@k` vs. `pass^k`

Because LLM generation is stochastic, running the exact same prompt and tool sequence k times can produce different trajectories and outcomes. Picking the wrong multi-trial metric can make a brittle system look production-ready on paper while failing constantly for real users.

Mathematical & Operational Contrast

Let $p \in [0, 1]$ be the single-trial probability that an agent or model completes a task correctly on one independent run, and let k be the number of independent trials.

$$\text{pass@k} = 1 - (1 - p)^k \quad \text{vs.} \quad \text{pass}^k = p^k$$

Metric	Formula (for single-run success p)	Meaning	Curve as k Increases	When to Use	When It Is Dangerous
`pass@k` *(At-Least-One Success)*	$1 - (1 - p)^k$ *(unbiased estimator over $n \geq k$ samples)*	Chance of seeing **one or more successes** across k independent runs.	**Rises toward 100%** (masks underlying flakiness)	Code generation with a compiler/unit-test verifier; swarm tournament prototyping where a deterministic filter picks the single working output.	**Customer-facing chat or autonomous agents:** users only see 1 run, and side-effecting tools cannot be casually run 5 times hoping one works.
`pass^k` *(Every-Run Consistency)*	p^k	Chance that **every single run across k trials** succeeds without failure.	**Decays exponentially toward 0%** (exposes variance)	**Customer-facing agents**, unattended background automation, regulated workflows, and mutating tool pipelines where every invocation must be reliable.	Over-optimizing `pass^k` on purely creative brainstorming tasks where diversity across samples is desirable.

The Exponential Decay of Consistency (`pass^k` Table)

Notice how a single-run accuracy of p = 0.90 looks impressive on `pass@5` (99.999%) while collapsing to 59.0% on `pass^5` and 34.9% on `pass^10` :

Single-Run Accuracy (p)	`pass@1` / `pass^1`	`pass@5` ($1 - (1 - p)^5$)	`pass^5` (p^5)	`pass^10` (p^{10})	Operational Interpretation
0.70 (70%)	70.0%	99.8%	**16.8%**	**2.8%**	Usable for 5-sample code synthesis with unit tests; **unusable** for live automation.
0.85 (85%)	85.0%	99.99%	**44.4%**	**19.7%**	Fails more than half the time over 5 consecutive runs.
0.90 (90%)	90.0%	99.999%	**59.0%**	**34.9%**	Still fails **41% of 5-run consistency suites**.
0.95 (95%)	95.0%	>99.999%	**77.4%**	**59.9%**	Minimum baseline before adding deterministic guardrails.
0.99 (99%)	99.0%	>99.999%	**95.1%**	**90.4%**	Production-grade step reliability (achieved via code gates + schema constraints).

Compounding Multi-Step Accuracy Across Sequential Agent Steps

The exact same multiplicative law governs a single execution of a **multi-step sequential workflow**. If an agent must complete M sequential steps (e.g., Classify Intent → Formulate Query → Select Tool → Parse Payload → Synthesize Final Answer) and each step has independent success probability p_i, end-to-end workflow success is the product of every step:

$$P(\text{Workflow Success}) = \prod_{i=1}^{M} p_i \quad (\text{or } p^M \text{ if each step has accuracy } p)$$

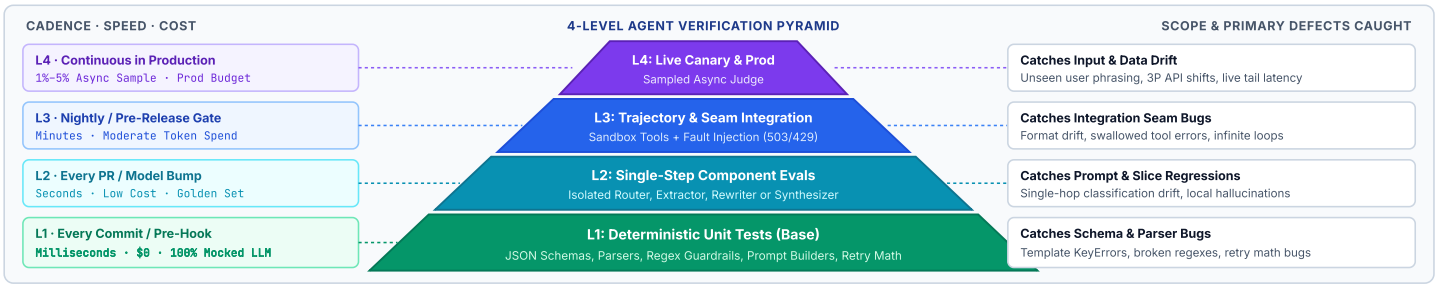
- **Example:** A 6-step autonomous workflow where every individual step scores **92%** in isolation (p = 0.92) has an end-to-end first-pass success rate of $0.92^6 \approx$ **60.6%**—meaning nearly **4 out of 10 production runs fail**.
- **Architectural remedies to beat compounding decay:**
  1. **Replace probabilistic steps with deterministic workflow nodes:** If steps 2 and 4 can be written as standard code, ADK workflow edges, or strict schema validators (p = 1.00), a 6-step pipeline shrinks to 3 probabilistic hops ($0.92^3 \approx 77.9\%$).
  2. **Insert programmatic verification gates between steps:** Catch and auto-correct malformed tool arguments or schema errors at step i before passing corrupt state to step i+1.
  3. **Bound step count:** Keep autonomous loops shallow and route predefined sequences through deterministic workflow orchestrators (`SequentialAgent`) rather than LLM routers.

EXAM & PRODUCTION RULE

Whenever a scenario involves an unattended workflow, customer-facing support agent, or financial/database tool execution, evaluate readiness using **`pass^k`** (every trial succeeds) and **end-to-end trajectory completion**, never `pass@k` . Use `pass@k` only when a downstream deterministic verifier (such as a test suite or compiler) can automatically filter k candidates before side effects occur.

4. The 4-Level Agent Test Pyramid & Integration Seams

Testing an agentic system exclusively through end-to-end live conversations is slow, expensive, and non-diagnostic; testing it exclusively through isolated unit tests misses catastrophic multi-step handoff failures. Production teams structure verification as a **4-Level Agent Test Pyramid**.



The 4 Levels in Detail

Level	Scope & Target	What Is Mocked vs. Live	Execution Cadence & Cost	What It Catches
Level 1: Deterministic Unit & Replay Tests	Tool JSON schemas, Pydantic validators, prompt template rendering, PII scrubbers, retry state machines, `adk conformance test --mode=replay`	**LLM is 100% mocked / replayed** from recorded fixtures. Pure code execution.	Every save / pre-commit hook · **Milliseconds to <5s · $0**	Broken schema definitions, template `KeyError` bugs, callback regressions, broken `state_delta` routing.
Level 2: Single-Step Component Evals	A single LLM call in isolation: intent router, entity extractor, query rewriter, or RAG answer generator	**Live LLM call** against a frozen, curated golden input + fixed context snapshot.	Every PR touching prompts/models · **Seconds to minutes · Low cost**	Prompt regressions, classification drift, hallucination on specific golden slices, schema compliance drops.
Level 3: Trajectory & Seam Integration Tests	Multi-turn agent loop across 2–8 steps: tool selection, argument passing, error recovery, subagent handoffs	**Live LLM + Environment Simulation / sandboxed test doubles** (including injected `503` , `429` , and empty-result fixtures).	Pre-release gate / nightly build · **Minutes · Moderate cost**	**Integration seam failures**, infinite tool loops, silent error swallowing, oversized tool payloads, state loss during compaction.
Level 4: Live Canary & Online Monitors	Real production traffic across the full deployed stack	**100% live** (canary slice, shadow pipeline, or 1–5% sampled **Online Evaluation Monitors**).	Continuous in production · **Budgeted % of prod spend**	Input/data drift, unpredicted user phrasing, latency spikes, real third-party API schema changes.

Why Integration Seams Hide Silent Bugs

An **integration seam** is the boundary where the output of Component A becomes the input of Component B (for example: *Query Rewriter* → *Hybrid Retriever*, *Tool Execution Output* → *Agent Context*, or *Planner Subagent* → *Executor Subagent*).

Compound Error Cascades across Multi-Step Pipelines: Every individual component can score **>95% in isolated component evaluations** while the end-to-end system fails on **>40% of real tasks** because independent step probabilities compound multiplicatively ($0.95^5 \approx 77\%$) and isolated evaluations feed idealized, clean inputs to each stage rather than handling noisy upstream artifacts.

Seam Failure Pattern	How Component A Behaves ("Passes L2")	How Component B Breaks Across the Seam	Architectural Fix
Format & Schema Drift	Extractor outputs dates as `"Oct 26, 2026"` or wraps JSON in ```` ```json ```` markdown fences (passes loose L2 check).	Downstream SQL/API tool expects strict ISO-8601 `2026-10-26` or raw JSON and throws a validation/parse error.	Enforce **strict structured outputs** (`output_schema` / JSON Schema) at the seam and validate with a Level 1/3 contract test.

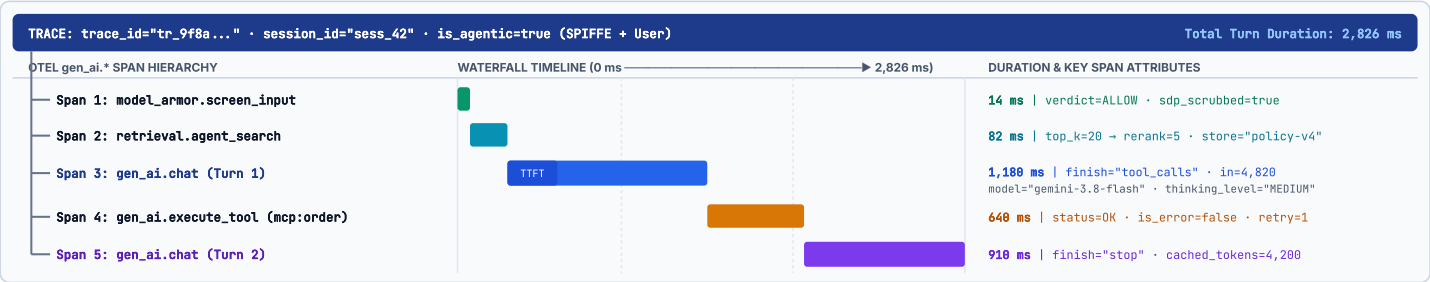
Seam Failure Pattern	How Component A Behaves ("Passes L2")	How Component B Breaks Across the Seam	Architectural Fix
Unbounded Tool Output & Silent Truncation	Search tool dumps a 40,000-token raw log into context (diluting model attention), or middleware naively slices the first 4,000 chars.	Either attention degrades across turns (**context rot**) or the stack trace at the bottom is cut off, causing a confident *"No errors found"*.	Truncate structurally (head + tail + error lines, or return pagination metadata `truncated: true, next_cursor: "..."`).
State Loss During Context Compaction	Sliding-window summarizer compresses 12 prior turns into a 3-sentence prose paragraph.	Summarizer drops an exact UUID, config flag, or constraint read in Turn 2 that Turn 14 needs to execute a tool safely.	Persist exact IDs, constraints, and slot values in structured **Session `state`** (`session.state`), never solely in summarized prose.
Swallowed Tool Failures (Empty vs. Error)	Tool wrapper catches a database timeout (`504`) and returns `[]` or `""` so the pipeline doesn't crash.	The model interprets `[]` as ground truth (*"You have zero open orders"*) instead of a transient system outage.	Never conflate empty results with failures; return explicit structured error flags (`is_error: true`, error code, retryability).
Distribution Shift Between Stages	Query rewriter expands short user queries into 80-word boolean strings that were never in the retriever's golden set.	Dense embedding model loses specificity on 80-word keyword-stuffed queries, dropping top-5 recall by 30%.	Evaluate *Query Rewriter + Retriever* together as a coupled pair in Level 3 seam tests using end-to-end Recall@K.

5. Observability, Distributed Tracing & Drift Detection

In a production agentic workflow, a single user message can fan out into Model Armor screening, hybrid vector retrieval, cross-encoder reranking, multi-turn `gemini-4-argon` or Claude reasoning calls, MCP tool executions, and A2A sub-agent delegations. Without standardized **distributed tracing**, diagnosing a 19-second stall, a reasoning loop, or a hallucination is guesswork.

OpenTelemetry `gen_ai.*` Semantic Conventions & Agent Observability

Instrument every workflow with **OpenTelemetry (OTel) Semantic Conventions for Generative AI** (`gen_ai.*` attributes across `invoke_agent`, `chat`, and `execute_tool` spans) so traces render uniformly across **Google Cloud Trace**, **Agent Registry Observability**, and vendor-neutral APMs:



Automatic Cloud Monitoring Metrics vs. OpenTelemetry `gen_ai.*` & Log-Based Metrics

Infrastructure and publisher-model telemetry arrive out of the box in **Cloud Monitoring (`GA`)** with zero code changes, whereas agent-level tool, session, loop-depth, and policy signals require **ADK OpenTelemetry semantic conventions** (`OTEL_SEMCONV_STABILITY_OPT_IN='gen_ai_latest_experimental'`) or **Cloud Logging log-based metrics**:

Telemetry boundary	Specific metric types / attributes	Setup & cardinality rule
Automatic Cloud Monitoring (`GA`, zero code changes)	• **Cloud Run:** `run.googleapis.com/request_count`, `run.googleapis.com/request_latencies`, `run.googleapis.com/container/billable_instance_time`, `cpu/utilizations`, `memory/utilizations`. • **GKE:** `kubernetes.io/container/*` (CPU, memory, storage, `restart_count`). • **Agent Runtime & Agent Gateway:** request throughput, RPC/HTTP status errors, container CPU/memory usage, and `p50` / `p95` / `p99` runtime/gateway latency. • **Model Platform Online Serving:** `aiplatform.googleapis.com/publisher/online_serving/token_count` and `aiplatform.googleapis.com/publisher/online_serving/model_invocation_latencies`.	Emitted automatically by Google Cloud control planes; tracks aggregate endpoint health and total project TPM/RPM, **not** which tool or session caused a spike.
OTel GenAI (`gen_ai.*`) & Log-Based Metrics *(explicit opt-in)*	• **Per-tool error rate:** `gen_ai.operation.name="execute_tool"` + `gen_ai.tool.name` + `status=ERROR` (`error.type`). • **Per-session token spend:** `gen_ai.conversation.id` + `gen_ai.usage.input_tokens` / `gen_ai.usage.output_tokens` (counting `thinking_tokens` + A2A/subagent fan-out). • **Reasoning loop depth:** child span count (`call_llm` + `execute_tool`) per root `trace_id`. • **Policy & guardrail denials:** Agent Gateway request logs (`resource.type="networkservices.googleapis.com/Gateway"` with `jsonPayload.authzPolicyInfo.policies.result="DENIED"` / `httpRequest.status=403`), IAP Data Access audit logs (`protoPayload.serviceName="iap.googleapis.com"` with `protoPayload.authorizationInfo.granted=false` on `iap.resources.egressViaIAP`), and Model Armor blocks (`SanitizeOperationLogEntry`).	Enable ADK OTel (`--otel_to_cloud`, `GOOGLE_CLOUD_AGENT_ENGINE_ENABLE_TELEMETRY=true`, `AdkApp(..., enable_tracing=True)`, or `get_gcp_exporters(enable_cloud_tracing=True)` on Cloud Run/GKE; GKE also needs an Agent Registry entry for Topology). **Cardinality rule: never** attach `session_id`, `user_id`, or `trace_id` as a Cloud Monitoring metric label!

The writer identity (usually the runtime service account) needs `roles/telemetry.tracesWriter`, `roles/logging.logWriter` and `roles/monitoring.metricWriter`. The ADK instrumentation guide's recommended environment for content-safe tracing:

```
GOOGLE_CLOUD_AGENT_ENGINE_ENABLE_TELEMETRY=true                # Agent Runtime only
OTEL_SEMCONV_STABILITY_OPT_IN=gen_ai_latest_experimental       # latest gen_ai conventions
OTEL_INSTRUMENTATION_GENAI_CAPTURE_MESSAGE_CONTENT=EVENT_ONLY  # messages as events, never "true"
ADK_CAPTURE_MESSAGE_CONTENT_IN_SPANS=false                     # no prompt text or PII in span attributes
```

- **High-cardinality metric label trap:** attaching unbounded `session_id` (`gen_ai.conversation.id`), `user_id`, or `trace_id` as a **Cloud Monitoring metric label** explodes active time-series cardinality, exhausts custom metric quotas, and spikes observability billing. Group Cloud Monitoring metrics only by bounded labels (`gen_ai.tool.name`, `model`, `status`), and query per-session or per-user breakdowns in **Cloud Logging Log Analytics**, **BigQuery log sinks**, or **Cloud Trace**.
- **Turning tracing on is a data decision:** once built-in instrumentation is enabled, text prompts and agent responses are sent to your project. Review it with the data owner, not only the SRE team.
- **Spans hold metadata, not conversations:** Agent Platform keeps latency, status codes and structure in spans and routes prompts and responses to Cloud Storage (recommended: no truncation, multimodal payloads, lifecycle rules) or Cloud Logging (text only, size limits). Because the two stores are separate, IAM can grant `roles/cloudtrace.user` for debugging without granting read access to conversation content.
- **`EVENT_ONLY`, never `true`:** with the latest `gen_ai` conventions, `OTEL_INSTRUMENTATION_GENAI_CAPTURE_MESSAGE_CONTENT=true` is an invalid configuration and no log or trace data is collected at all.
- **Media offload:** for images and large documents, set `OTEL_INSTRUMENTATION_GENAI_COMPLETION_HOOK=upload`, `OTEL_INSTRUMENTATION_GENAI_UPLOAD_FORMAT=jsonl` and `OTEL_INSTRUMENTATION_GENAI_UPLOAD_BASE_PATH=gs://...` so payloads land in a bucket instead of inside spans.
- **Online monitors depend on captured content:** they read `gen_ai.input.messages`, `gen_ai.output.messages`, `gen_ai.system_instructions` and `gen_ai.tool.definitions` from the inference event. Switching content capture off also switches off online evaluation, so address PII by redacting upstream (**Sensitive Data Protection** de-identification templates, **Model Armor**) and by restricting the log bucket (`roles/logging.privateLogViewer`) or Cloud Storage bucket.
- **Dual-Attribution Agentic Audit Logging (`is_agentic=true`):** When an agent acts on behalf of a human user, Cloud Audit Logs and execution logs tag events with `is_agentic=true` and record **both** (1) the cryptographic SPIFFE **Agent Identity** (`principal://...` / `spiffe://...`) and (2) the initiating **human user principal**, which removes "confused deputy" blind spots in forensics.

Reading one agent turn in a trace

The agent's Traces tab offers a Session view (multi-turn conversations with session token totals and evaluation scores), a Trace view (one end-to-end request) and a Span view (individual model calls, API requests and tool executions). Open one trace for a single turn and walk it top-down:

Order	Span to inspect	Healthy pattern	Investigate when
1	Root `invoke_agent`	One trace per user turn; total duration roughly equals the sum of child spans	A gap before the first child (cold start or blocking initialization), or one turn split across several traces (context propagation broken)
2	LLM spans (`call_llm` wrapping `generate_content {model}`)	A small, stable number of calls per turn; input tokens grow modestly turn over turn; each call ends in a final answer or a well-formed tool call	Input tokens climbing every turn (history bloat), output cut off at the token limit, repeated retries after `429`
3	Tool spans (`execute_tool`; MCP calls carry `mcp.method.name = tools/call` and `gen_ai.tool.name`)	Independent tools overlap in time; compact results; no `error.type`	Same tool with the same arguments three or more times in a row, `error.type` set, or a serial waterfall of multi-second calls
4	Sub-agent handoffs (nested `invoke_agent` spans)	A shallow parent-to-specialist tree that matches the intended design and the Topology tab's edges	A to B to A ping-pong, unexpected specialists, or depth that grows with every turn
5	Retrieval spans (search and RAG tools appear as `execute_tool`)	Fast, non-empty results whose content shows up in the final answer	Zero or irrelevant results followed by a confident answer (hallucination risk)
6	Guardrail spans (Model Armor, Preview)	Screening adds little latency and allows benign traffic	An empty or refused answer that a Model Armor interception explains, which rules out a retrieval miss
7	Evaluation tab on the trace	Online monitor scores and rationales agree with what you just read	Low scores on a trace that looks clean (check the rubric) or high scores on a broken one (check the autorater)

Troubleshooting Production Agent Issues via Telemetry Views

Agent Registry's Observability tab (Overview, Evaluation, Models, Tools, Usage, Logs) plus the agent's Traces and Topology tabs map production symptoms to root causes. The Models view isolates p95 latency, errors, quota failures and tokens per model; the Tools view shows p95 latency, call counts and error rates per tool, plus how often a turn called no tool at all.

Symptom	Likely cause	Where to look	Fix
Tool latency spike (turn `p95` jumps while LLM time stays flat)	Unpooled database connections, cross-region hops, or independent reads executed one after another	Tools view `p95` per tool; Traces waterfall with long or serial `execute_tool` spans	Co-locate runtime and data, use **MCP Toolbox** connection pooling, fan out independent reads (`ParallelAgent`), set per-tool timeouts, cache hot lookups
Reasoning or tool-call loops	Ambiguous tool errors (an empty string instead of actionable feedback) or cyclic sub-agent routing	Traces: 8 or more consecutive `call_llm` and `execute_tool` pairs with identical or oscillating arguments; Topology for A to B to A edges	Return structured error hints (`ReflectAndRetryToolPlugin`), cap iterations (`max_llm_calls` , `max_iterations`), break cycles with deterministic state flags (per-cause fixes in the runaway loop triage table below)
Ignored tools ("no-tool interactions")	Vague or overlapping tool descriptions, so the model answers from parametric memory	Tools view no-tool rate alongside rising `HALLUCINATION` scores on domain questions	Sharpen tool descriptions and schemas; add `trajectory_in_order_match` CI gates
Prefill or thinking bottleneck	`thinking_level="HIGH"` on routine hops, or volatile text at the top of the prompt defeating prefix caching	Models and Traces views: `thinking_tokens` dominating generation time, high TTFT with `cached_tokens = 0`	Route simple steps to `gemini-3.5-flash-lite` (`minimal`) or `gemini-3.8-flash` (`low`); move volatile state after `static_instruction`
Rising token cost	Unbounded conversation history, large tool payloads, many tool schemas sent on every turn	Overview token usage (input vs. output); per-span input tokens growing turn over turn	ADK context compaction (token-based or sliding window), context caching, pruned tool results, tools and skills loaded on demand
Hallucinated answers	Empty or weak grounding, or the agent answering despite an empty retrieval	Evaluation view hallucination rate; retrieval span output compared with the final response	Ground on Agent Search or RAG Engine, add reranking, instruct the agent to abstain when retrieval returns nothing
Retrieval misses	Dense-only search missing exact codes and acronyms, or chunks stripped of their section headings	Retrieval `execute_tool` spans returning irrelevant or zero results	Hybrid (dense plus keyword) retrieval, chunking that keeps ancestor headings, reranking, corpus refresh
Quality drift after a model or prompt change	New model version or prompt edit changes tool selection, formatting or refusal behavior	Evaluation dashboard trends across agent versions; Automatic Loss Analysis clusters for the new version	Roll back to the prior pinned version, add failing traces to the golden set, re-run the optimizer before re-promoting (see `#feedback-flywheel`)
`429` / quota errors	Burst traffic or runaway loops exhausting model quota (`RESOURCE_EXHAUSTED`)	Models view quota failures; Cloud Logging entries with `429`	Full-jitter backoff (see `#resilience-stack`), loop caps, concurrency limits, Provisioned Throughput for predictable capacity
Cold starts	No idle instance when a request arrives; heavy imports and client setup at startup	Gap before the first child span on the first turn; instance count rising during spikes	Raise `min_instances` (Agent Runtime) or `--min-instances` (Cloud Run) to cover baseline load, lazy-load heavy libraries, slim images; for async ADK agents on Agent Runtime start `container_concurrency` at a multiple of 9 (for example 36) and watch memory
Traces tab or online monitor empty	Telemetry never enabled, or content capture disabled	Cloud Trace for `gen_ai.*` attributes; Logs Explorer filtered on `resource.labels.online_evaluator`	Apply the opt-in settings above; confirm monitor filters with Initial Inspection

Runaway loop triage

The loop row above names the shared symptom. Five different causes produce it, and each one needs its own fix. The common rule: a loop must never end on model judgment alone. An iteration cap, a terminal route or a budget has to end it.

Loop cause	Trace signal	Deterministic fix
`LoopAgent` with no exit path	The same child agent spans repeat in the same order until something external stops the run; `LoopAgent` never decides to stop by itself	Set `max_iterations` on every `LoopAgent` and give the checker an exit tool that sets `tool_context.actions.escalate = True` (a custom agent can instead yield an event with `Event Actions(escalate=True)`)
Graph back-edge with no terminal branch	Two graph nodes alternate in the trace and the run ends only on a timeout or budget	Keep an iteration counter in state and have the router node emit a terminal route (for example `DONE`) once the counter reaches its cap; point `DEFAULT_ROUTE` , which fires when no other route on that node matches, at a terminal or fallback node rather than at the back-edge
Tool error retried forever	`execute_tool` spans with `error.type` set, each followed by an LLM call that reissues the same tool with the same arguments	`ReflectAndRetryToolPlugin(max_retries=...)` caps retries per tool (default 3, tracked per invocation) and returns structured guidance to the model; for errors that cannot self-heal (`401` , `403` , `404`), a `before_tool_callback` that counts calls per tool-and-arguments key returns a dict once the limit is hit, which skips the tool and hands the model a "do not retry" result
A2A ping-pong delegation	Nested spans alternate between two remote agents inside one distributed trace; Topology shows edges in both directions	Keep delegation acyclic (a specialist never delegates back to its caller), carry a hop or depth counter in the delegated task and refuse past a fixed depth, and give each task a deadline and token budget
Over-strict critic	Every run uses the full `max_iterations` even on drafts a reviewer would accept; the escalate event never appears	Calibrate the critic rubric against golden examples labeled acceptable, and when the cap is hit return the best draft flagged as partial or hand off to a human instead of failing silently

- **Minimal exit wiring:** give the critic an exit tool that sets `tool_context.actions.escalate = True` when the draft passes, and always construct the loop with a hard cap, for example `Loop Agent(name="refine", sub_agents=[critic, refiner], max_iterations=5)`, so the loop stops even if escalation never fires.
- **Agent Anomaly Detection (Preview, allowlisted):** an Agent Platform service that analyzes ADK OpenTelemetry traces and logs out of band, so it adds no latency to the agent. Its agentic cascading failures category (OWASP ASI08) covers infinite execution loops and oscillating retries, and a separate resource exhaustion category covers runaway token or compute use. Findings appear in Security Command Center and detection metrics in Cloud Monitoring. It requires ADK for Python 2.1.0 or later on Agent Runtime, Cloud Run or GKE, with tracing on and prompt and response content captured.
- **Detection is not prevention:** anomaly findings arrive after the loop has already spent tokens, so the caps and terminal routes in the table stay mandatory.
- **Anti-pattern (raising the cap):** bumping `max_iterations` or `max_llm_calls` so the agent "has room to finish" leaves the missing exit signal in place and makes each runaway cost more.
- **Anti-pattern (default route as back-edge):** mapping `DEFAULT_ROUTE` to the loop's back-edge means any unexpected route value, including a typo, re-enters the cycle.

SLIs, SLOs & Two Distinct Forms of Production Drift

Production AI systems track **four SLI dimensions**: **Responsiveness** (`p95 TTFT` , `p95` turn latency), **Pipeline Reliability** (`429` / `5xx` rate, tool `is_error` rate, fallback rate, unexpected `max_tokens` / `refusal` stop reasons), **Economic Efficiency** (cache hit ratio >75%, cost per resolved task, `p95` turns/session), and **Semantic Quality** (sampled **Online Evaluation Monitor** pass rates ≥ 96% and human escalation rate).

Even with zero code deploys, quality degrades over time through two distinct drift mechanisms with opposite fixes:

Drift Type	Root Cause	Symptoms in Telemetry	Engineering Remediation
1. Input / Data Drift *(World Changes)*	Real user queries, new product SKUs, updated regulations, or changed third-party API payloads diverge from your static golden set and RAG corpus.	Model ID and prompt hash are unchanged and infra SLIs are green, yet **retrieval similarity scores drop**, out-of-domain queries rise, and Online Monitor faithfulness declines gradually.	**1.** Monitor query embedding distribution shift & retrieval score histograms. **2.** Refresh the RAG / Agent Search corpus. **3.** Feed SDP-scrubbed failing traces into the **LM Feedback Flywheel** to expand the golden evalset.
2. Model / Provider Drift *(System Changes)*	Calling a moving model alias (e.g., `-latest`), an unannounced provider routing update, or an unvalidated prompt/tool-schema edit.	Sudden step-change shift in output length, `refusal` rate, JSON parse errors, or tool-calling aggressiveness across all topics simultaneously.	**1. Pin exact immutable model snapshot IDs** and Agent Registry versions. **2.** Run automated Level 2/3 eval gates before any model cutover (using **Data-Driven Prompt Optimizer** when migrating model families). **3.** Maintain instant config rollback to the prior pinned ID.

5-Target Alerting Policy Matrix & Multi-Window SLO Burn-Rate Alerts

Configure operational alerts across five high-signal agent boundaries while guarding against time-series cardinality blowups and judge-token billing spikes:

Alert Target	Telemetry Source & Aggregation	Example Condition (Directional)	Critical Guardrail / Anti-Pattern
1. Tool Call Error Rate	OTel `execute_tool` spans (`error.type`) or log-based counter grouped by `gen_ai.tool.name`	`> 5%` error ratio over a rolling 5-minute window per `gen_ai.tool.name`	**Never** attach high-cardinality `session_id` or `user_id` labels to the Cloud Monitoring metric descriptor.
2. `p95` Tool & Turn Latency	`run.googleapis.com/request_latencies` , `model_invocation_latencies` , or tool duration histograms with **`ALIGN_PERCENTILE_95`**	`ALIGN_PERCENTILE_95` exceeds latency SLO (e.g., `8 s` interactive chat) for 10 minutes	**Never alert on arithmetic mean latency**, which masks heavy-tail reasoning and retry stalls behind fast cache hits.
3. Runaway Loop Depth	Child span count per `trace_id` (`steps_per_turn > max_iterations - 1`) + **Agent Anomaly Detection (`Preview` , OWASP ASI08)** in SCC	Spike in turns hitting `max_iterations - 1` or any ASI08 cascading-failure finding	Out-of-band anomaly detection fires *after* tokens burn; always pair with an in-band hard `max_iterations` / `max_llm_calls` cap.
4. Token Spend per Session	Cumulative `input_tokens + output_tokens + thinking_tokens` per `gen_ai.conversation.id` via **Log Analytics / BigQuery sink**	`p99` tokens/session exceeds `2×` trailing 7-day baseline (counting subagent fan-out)	Compute per-session `p99` in Log Analytics or BigQuery, never as a per-session Cloud Monitoring time series.
5. Online Monitor Quality Drop	`aiplatform.googleapis.com/online_evaluator/scores` filtered by `metric.labels.evaluation_metric_name`	Rolling 30-min mean score `< 0.85` (or `< 0.80` default template) on `HALLUCINATION` / `TOOL_USE_QUALITY`	Apply **cost-control trace sampling filters** (`duration > 5s` or **`5%` random sample** + max samples/run); **never sample 100%** of high-QPS traffic (5–31 judge calls/trace, e.g., 31 calls on `SAFETY` `safety_v3` !).

(Also watch sustained `429` quota failures in the Models view and `>50%` jumps in the Tools view "no-tool interaction" rate.)

Multi-Window SLO Error-Budget Burn-Rate Alerts: Static 5-minute threshold alerts either page on-call engineers for harmless 30-second upstream blips or fail to fire during a slow `0.3%` error leak that silently exhausts a 30-day `99.9%` SLO. Pair a **1-hour `10×` fast-burn alert** (pages immediately on acute outages) with a **24-hour `2×` slow-burn alert** (creates a ticket on gradual error-budget leaks) using Cloud Monitoring's native SLO selector:

```
select_slo_burn_rate("projects/<proj>/services/<svc>/serviceLevelObjectives/<slo>", "3600s") > 10.0
select_slo_burn_rate("projects/<proj>/services/<svc>/serviceLevelObjectives/<slo>", "86400s") > 2.0
```

Directional thresholds: numeric cutoffs (`5%` , `8 s` , `0.85` , `10.0` , `2.0`) are SRE baselines, not fixed platform limits. Calibrate thresholds against your service's error budget and two to four weeks of production traces.

Online monitors and quality alerts: catching drift

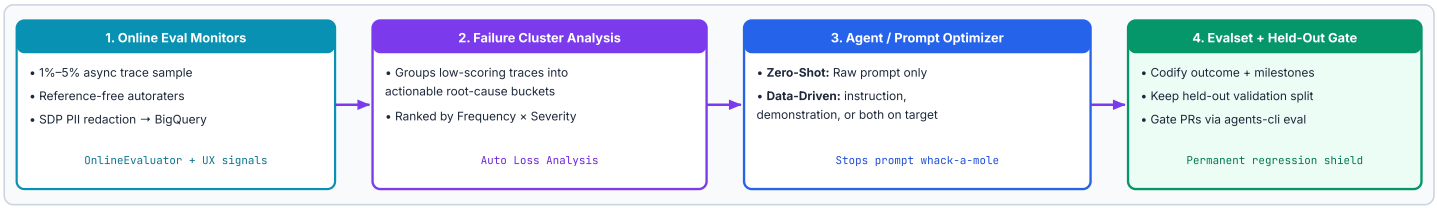
Offline evals prove a release was good on the day it shipped. **Online Monitors** keep scoring production traffic afterwards, and **quality alerts** turn a falling score into a page, so both drift types from the table above surface without a manual spot check.

1. **Instrument for evaluation:** enable Cloud Trace export and content capture as events (`OTEL_SEMCONV_STABILITY_OPT_IN=gen_ai_latest_experimental`, `OTEL_INSTRUMENTATION_GENAI_CAPTURE_MESSAGE_CONTENT=EVENT_ONLY`) so each trace carries `gen_ai.agent.name`, `gen_ai.conversation.id` and the inference messages.
2. **Create a monitor:** on the Online monitors tab of the Evaluation page, pick the agent, evaluate all traces or a filtered slice (for example by duration or token usage), choose metrics such as safety, task success, tool use quality or hallucination, then set a sampling percentage and a maximum samples per run to cap judge cost.
3. **Scheduled scoring loop:** roughly every 10 minutes the monitor queries sampled traces from Cloud Trace and Cloud Logging, scores them with the Agent Platform evaluation service, writes results to Cloud Logging and exports numeric scores to Cloud Monitoring.
4. **Alert on the scores:** from the monitor's menu, choose **Create alerting policy** to get one template per configured metric, or use **Recommended Alerts** on the agent's Evaluation dashboard (an aggregate low-score template plus per-metric templates such as task success and tool use quality). Route incidents to Slack, email or Pub/Sub; each incident carries the monitor's labels so responders land on the traces that caused it.
5. **Triage the drop:** compare score trends across agent versions on the Evaluation dashboard, open the Evaluation tab of low-scoring traces for rubric verdicts and rationales, then run **Automatic Loss Analysis** to classify rubric failures into loss patterns and semantic clusters.
6. **Classify and fix:** a step change aligned with a deploy or model switch points to model or prompt drift (roll back, then fix through the flywheel). A gradual decline with no deploy points to input drift (refresh the corpus, expand the golden set). Either way, the clustered failures feed the closed loop in `#feedback-flywheel`.

- **Access control:** online evaluation runs under the project-level service agent (P4SA), and anyone allowed to create an `OnlineEvaluator` can attach it to any agent in the project. Restrict that permission to authorized administrators.
- **Silent monitor:** when a monitor is active but shows no results, confirm live traces contain `gen_ai.*` attributes, re-check filters with Initial Inspection, and search Logs Explorer for `resource.labels.online_evaluator` error entries.

6. LM Feedback Flywheel, Failure Clustering & Agent Optimizer

Drift detection tells you *that* quality moved; the **LM Feedback Flywheel** and **Agent Optimizer** ensure every production failure class is discovered automatically, grouped by root cause, fixed programmatically without "prompt whack-a-mole," and permanently guarded against in CI/CD.



Step-by-Step Closed-Loop Feedback Flywheel

1. **Capture Multi-Source Signals:** Collect explicit user feedback (thumbs-down, bug reports), implicit behavioral friction (immediate re-prompting, session abandonment, human escalation), and automated **Online Evaluation Monitors (`OnlineEvaluator`)** that continuously sample 1–5% of live Cloud Trace spans and score them asynchronously with reference-free autoraters (`HALLUCINATION`, `TOOL_USE_QUALITY`, `GENERAL_QUALITY`).
   - *IAM Governance Rule:* Online monitors execute under a platform service agent that reads trace payloads and spends judge tokens—restrict monitor creation/modification IAM permissions to authorized AgentOps engineers.
2. **Scrub Before Storage:** Export flagged `trace_id` records to BigQuery via Cloud Logging sinks after passing payloads through **Sensitive Data Protection (SDP)** de-identification templates (`ReplaceWithInfoTypeConfig` or `CryptoDeterministicConfig` surrogates) so promoted `evalset` cases and **Example Store** few-shot pairs never leak customer PII or credentials into Git repos or prompts.
3. **Triage via Failure Cluster Analysis (Automatic Loss Analysis):** Instead of playing whack-a-mole on isolated support tickets, invoke **`client.evals.generate_loss_clusters()`** (or `agents-cli eval analyze`) over low-scoring online traces and failed simulation runs. An analyzer LLM clusters raw failures into actionable root-cause buckets (e.g., *"Cluster #1 (34% of failures): `book_flight` receives relative dates like 'next Friday' instead of ISO-8601"*, *"Cluster #2 (19%): premature final answer before reading page 2 of search results"*) ranked by Frequency × Severity.
4. **Reproduce & Codify Outcomes (Not Brittle Transcripts):** Convert representative traces from top clusters into reproducible `evalset` cases (`user_content`, `session_input`, `intermediate_data.tool_uses` / `reference_trajectory`) and **Example Store** exemplars. Assert **outcomes and required safety milestones** (`trajectory_in_order_match`) rather than exact word-for-word transcripts so valid alternative reasoning paths still pass.
5. **Protect Generalization with a Held-Out Split & Watch Saturation:** Keep a **held-out validation slice** (`train` vs. `validation`) invisible to both engineers and automated prompt optimizers (`adk optimize` / `optimize-agent`) during tuning to prevent prompt memorization/overfitting, and continuously refresh suites that saturate near 100%.

Programmatic Prompt Optimization (`Agent Optimizer` / Prompt Optimizer)

Manually rewriting a 4,000-token system prompt to fix Failure Cluster #1—or migrating prompts from an older model to `gemini-3.8-flash`, `gemini-4-argon`, or a smaller `Gemma 4` deployment—often introduces regressions on previously working behaviors. Agent Platform provides two automated optimization engines:

Optimizer Engine	Inputs Required	Optimization Modes & Mechanism	When to Choose
Zero-Shot Optimizer (`client.prompt_optimizer.optimize_prompt`)	**Only the raw prompt / system instruction** (zero labeled examples or eval dataset required).	`optimization_mode="zero_shot"` : Analyzes prompt structure against model best practices and immediately returns `suggested_prompt` plus `applicable_guidelines` (e.g., adding missing role boundaries, fixing ambiguous constraints, removing redundant instructions).	Cold-start prototyping, cleaning up ad-hoc developer prompts in seconds, or preparing an initial baseline before collecting golden data.
Data-Driven Optimizer & `Agent Optimizer` (`adk optimize` / `agents-cli eval optimize` / GEPA job)	System instructions + prompt template + **labeled/scored `train` dataset** (plus held-out `validation` split) + **Gen AI Evaluation metrics** + **target model ID**.	Runs an iterative closed-loop `optimize-agent` job (`evaluate → analyze failure clusters → mutate prompt/demos → re-evaluate on target model`) across three selectable modes: • **`instruction`** : Rewrites system instructions only. • **`demonstration`** : Selects optimal few-shot examples from candidates (or populates **Example Store**). • **`instruction_and_demo`** : Jointly optimizes instructions and few-shot demonstrations.	**1.** Fixing ranked **Failure Clusters** without regressing existing evals. **2. Model migrations** (adapting prompts to `gemini-4-argon` or distilling a frontier prompt down to `gemini-3.5-flash-lite` / `Gemma 4` for cost/latency savings).

7. Deployment Runtime Selection: Agent Runtime vs. Cloud Run vs. GKE

Selecting the wrong compute runtime in Google Cloud creates immediate architectural friction—either starving a stateful agent of memory, breaking MCP server hosting, or saddling a small team with unnecessary Kubernetes node management. Apply **Rule 0 (Managed by Default)**: start at the most managed runtime that satisfies your language, protocol, accelerator, and isolation constraints.

Google Cloud Agentic Runtime Decision Matrix

Architectural Dimension	Agent Runtime *(formerly Agent Engine)*	Cloud Run *(Services & Jobs)*	Google Kubernetes Engine (GKE)	Managed Agents API *(Preview)*
Primary Workload Fit	Stateful Python/ADK, LangGraph, CrewAI, or A2A orchestrators (`reasoningEngines`) needing turnkey agent ops	**Custom Streamable HTTP MCP servers** (`--functional-type=mcp-server`), polyglot agents (TS, Java, Go, Python), webhook/Pub/Sub event-driven agents	**Self-hosted open-weight models (`Gemma 4` , `Llama`) on GPUs/TPUs**, dense multi-agent meshes, strict kernel-isolated multi-tenant sandboxes	Declarative autonomous coding, research, and file-processing agents (`Agents API` + `Interactions API`)
Operational Overhead	**Lowest** (fully managed agent container lifecycle)	**Low** (serverless container-to-URL, scales to zero)	**Highest** (cluster, node pools, networking, controller upgrades)	**Zero** (Google runs the loop and no-egress Linux container)
State & Memory Model	**Built-in:** Native **Agent Platform Sessions**, **Memory Bank**, **Example Store**, and **Code Execution** sandbox	**Stateless / Ephemeral RAM:** Local memory vanishes on scale-in; must attach external **Sessions** (`agentengine://`), **Memorystore**, **Firestore**, or **Cloud SQL**	StatefulSets, **Agent Substrate** (`<1s` auto-snapshot/restore of memory & files on GKE Standard), or external managed DBs	Server-side state chained via `previous_interaction_id` and `environment_id` (`global` endpoint)
Scaling & Duration	Managed autoscaling up to **`1,000` instances** (**`100` in VPC-SC**); `min_instances` for zero cold start; up to **`6,000` revisions** (`traffic-split` bypassed if Agent Gateway routes); multi-day async workflows	Scales `0 → 1,000` rapidly; HTTP timeout up to **60 minutes**; Cloud Run Jobs up to **24h** for batch backfills	Granular HPA/VPA, custom queue metrics, GPU time-slicing, Spot VMs, **Committed Use Discounts (CUDs)**	Synchronous streaming or `background=True` multi-hour async execution
Hosts Custom MCP Servers?	**No** (consumes MCP/A2A, does not host standalone MCP servers)	**Yes — Primary serverless choice** for Streamable HTTP `/mcp` servers	**Yes** — High-density, stateful, or private in-cluster MCP fleets	**No** (supports server-side tools only)
Custom GPUs / Self-Hosted Models?	**No** (calls managed model APIs or private GKE/Model Garden endpoints over PSC)	**Limited** (serverless L4 / RTX PRO 6000 GPUs for lightweight bursts)	**Yes — Primary choice** for vLLM/TGI serving of **Gemma 4** on GPUs/TPUs *(never proprietary Gemini weights!)*	**No**
Code Sandbox Isolation	Managed **Code Execution** micro-container (`<1s` startup, 100 MB I/O, up to 14-day TTL) & **Computer Use** (`Preview`)	Call external Agent Platform Code Execution sandbox API	**Agent Sandbox on GKE** (`SandboxTemplate` + `SandboxWarmPool` pre-warmed **gVisor** pods; blocks `hostNetwork` / `hostPath`)	Built-in isolated Linux container (outbound network disabled by default)

Runtime dimensions in detail

The matrix above names the best fit. The table below compares the three general-purpose runtimes on the dimensions that usually decide an architecture review.

Dimension	Agent Runtime	Cloud Run	GKE
Managed agent primitives	Sessions, Memory Bank, Code Execution sandbox and Agent Identity available to the deployed agent; bidirectional streaming (`Gemini Live API`) supported	None built in; call Agent Platform Sessions or Memory Bank through the SDK, or bring Firestore, Memorystore, Spanner or Cloud SQL	None built in; wire state yourself or use Agent Sandbox and Agent Substrate
Scale-to-zero	`min_instances` defaults to 1, so one warm instance stays up	Scales to zero when idle unless you set minimum instances	Pods scale with HPA and nodes with the cluster autoscaler; the cluster itself stays provisioned
Cold start and concurrency	Raise `min_instances` to absorb baseline load; `container_concurrency` defaults to 9 (one per agent process), and async ADK agents should start at a multiple of 9 such as 36 (too high risks OOM); `max_instances` caps at `1,000` (`100` in VPC-SC)	Default 80 concurrent requests per instance for console deploys (80 × vCPU with gcloud or Terraform), configurable up to 1,000; minimum instances and startup CPU boost cut cold starts	Warm pods have no cold start; new pods pay image pull and readiness time, which you tune
Accelerators	None exposed; calls hosted models	NVIDIA L4 (24 GB) and NVIDIA RTX PRO 6000 Blackwell (96 GB) GPUs, with scale to zero; L4 needs at least 4 CPU and 16 GiB memory	Full GPU and Cloud TPU node pools for self-hosted models
Runtime contract	Custom containers listen on `0.0.0.0:8080`; `/api/reasoning_engine` (unary `query`) and `/api/stream_reasoning_engine` (`streamQuery`) are optional but required for SDK use, and the streaming one for the console playground; `AdkApp` is the reference implementation	Any container that serves on `$PORT` (HTTP, gRPC, WebSockets)	Standard Kubernetes Pod spec: sidecars, DaemonSets, StatefulSets
Sandboxing untrusted code	Managed Code Execution sandbox	Per-instance sandboxed execution environment; call the managed Code Execution sandbox for model-written code	GKE Sandbox (gVisor) and Agent Sandbox with warm pools
Networking	Private Service Connect interface (`networkAttachment`, `/28` min subnet) + `AgentConnectivityTemplate` (`vpc Egress: ALL_TRAFFIC`, DNS `restricted.googleapis.com` `199.36.153.4/30` under VPC-SC)	Direct VPC egress, internal ingress, VPC Service Controls, IAP	Native VPC subnets, private clusters, `Network Policy`, service mesh
Identity	Agent Identity (GA) per agent, or a service account	Per-service service account or Agent Identity; IAP for callers	Workload Identity Federation for GKE maps Kubernetes service accounts to IAM principals
Ops burden	Lowest: packaging, revisions and telemetry hooks managed	Low: no nodes, you own the container	Highest: cluster upgrades, node pools, networking, controllers
Billing shape	Compute and memory of running instances, including the warm baseline; see Agent Platform pricing	Request-based or instance-based billing; GPUs billed per GPU second	Cluster fee plus nodes (Standard) or requested Pod resources (Autopilot)

Directional cold-start figures: the Agent Runtime scaling guide shows one benchmark (300 concurrent requests, default `min_instances=1`) averaging about 4.7 seconds on a cold run versus about 0.4 seconds warm, and roughly 1.4 seconds once `min_instances=10`. Cloud Run documents about 5 seconds to start a GPU instance with drivers preinstalled. Treat these as examples and load test your own agent.

Choose this runtime when

- **Stateful ADK orchestrator that needs sessions, memory and per-agent identity with little glue code:** choose Agent Runtime.
- **Real-time voice or video agent that streams in both directions:** Agent Runtime bidirectional streaming fits, but it is Preview; confirm the launch stage is acceptable or host a WebSocket service on Cloud Run.
- **Custom Streamable HTTP MCP server or webhook tool that idles most of the day:** choose Cloud Run for scale to zero and IAP in front of the endpoint.
- **Polyglot agent or web front end packaged as a standard container:** choose Cloud Run.
- **Bursty small open model inference without running a cluster:** choose Cloud Run with an L4 or RTX PRO 6000 GPU.
- **Large self-hosted open-weight models on multi-GPU or TPU nodes inside a private VPC:** choose GKE.
- **Model-written or user-supplied code that must run with kernel isolation next to your workloads:** choose GKE with GKE Sandbox or Agent Sandbox; for short stateless snippets, the managed Code Execution sandbox is enough.
- **Workload that needs sidecars, DaemonSets or StatefulSet caches in one namespace:** choose GKE.
- **Anti-pattern:** picking GKE for a single stateless agent "for flexibility". You inherit cluster upgrades and on-call load with no capability you need.
- **Anti-pattern:** raising `container_concurrency` on Agent Runtime for a synchronous agent. The default of 9 assumes blocking code; only async agents benefit, and memory limits still apply.

GKE Sub-Architecture Choice: Agent Substrate vs. Agent Sandbox

When requirements mandate GKE, distinguish its two specialized agent orchestration architectures:

- **Agent Substrate on GKE (Actor-Based, GKE Standard):** Best for high-density stateful agents that frequently go idle between user turns. Active agents multiplex over a shared pool of warm Pods; idle agents automatically checkpoint memory and filesystem state to storage and restore in `< 1 second`, maximizing node utilization.

- **Agent Sandbox on GKE (Pod-Based, GKE Autopilot or Standard):** Best when each agent or untrusted code execution requires a dedicated Kubernetes Pod (`Sandbox` / `SandboxClaim` CRDs, `Sandbox WarmPool` for sub-second startup), full `PodSpec` sidecars, hardware accelerators (GPUs/TPUs), and mandatory **gVisor** kernel isolation enforced via Validating Admission Policy.

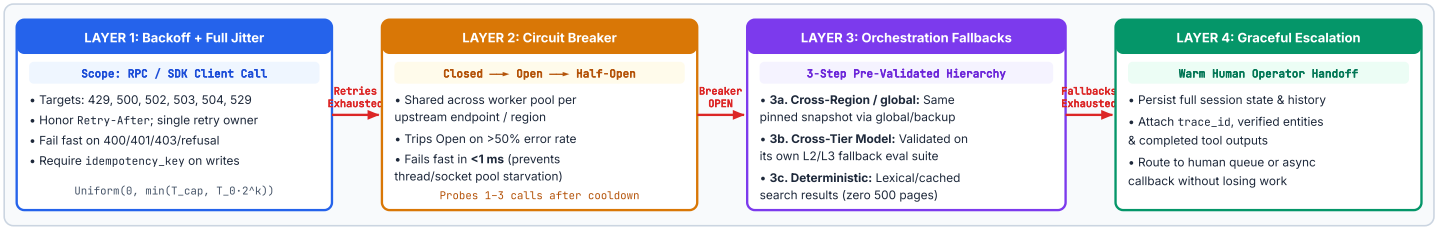
Runtime Capacity, Quota & Configuration Traps

- ***"Region temporarily at capacity"*** vs. `429 RESOURCE_EXHAUSTED` :
  - Seeing `"Region temporarily at capacity"` when deploying an Agent Runtime instance means **physical regional compute pool exhaustion**, *not* an IAM error or project quota cap—filing a quota increase ticket will not resolve it. Deploy to an alternate supported region, right-size vCPU/memory, or retry off-peak.
  - Seeing `429 RESOURCE_EXHAUSTED` during Gemini API inference indicates shared Standard PayGo contention or TPM/RPM limits. Route stateless requests to the `global` **endpoint**, apply full-jitter backoff, enable **Priority PayGo** for bursty interactive paths, or purchase **Provisioned Throughput (GSUs)** for predictable baseline load. *(Remember: Provisioned Throughput does not cover Batch Prediction or internal calls made by other managed services like Agent Search, even when using the same model.)*

- **Reserved Environment Variables on Agent Runtime:** Never override `GOOGLE_CLOUD_PROJECT` or `PORT` , and never define custom variables starting with the reserved `GOOGLE_CLOUD_AGENT_ENGINE*` prefix (except documented platform toggles like `GOOGLE_CLOUD_AGENT_ENGINE_ENABLE_TELEMETRY=true`).
- **Agent Registry Auto-Registration Rules:** Agents on **Agent Runtime**, **Gemini Enterprise**, official Google MCP servers, annotated **GKE** workloads, and **Cloud Run** services deployed with `--functional-type=agent` + `--identity-type=agent-identity` auto-register in **Agent Registry**; unflagged Cloud Run services and external/on-prem MCP servers must be registered via the writable `Service` resource.

8. Production Resilience Stack & Fault Isolation

LLM APIs and external tools operate under tight token-per-minute (TPM) and request-per-minute (RPM) quotas, variable compute load, and occasional regional brownouts. Production resilience requires a **4-layer defense stack** where each control lives at a distinct architectural boundary.



Layer 1: Exponential Backoff with Full Jitter & Idempotency

When an upstream model endpoint returns `429 Too Many Requests` or `529 Overloaded` , naive fixed-interval retries synchronize across all concurrent workers—creating a **retry storm (thundering herd)** that guarantees the next wave fails too.

1. **Honor `Retry-After` first:** If the response includes a `retry-after` or `retry-after-ms` header, sleep for at least that server-specified duration.
2. **Use Exponential Backoff with Full Jitter** when no header is present:

$$t_{sleep} = \text{Uniform}(0, \min(T_{cap}, T_{base} \cdot 2^{attempt}))$$

Randomizing the wait across [0, cap] desynchronizes concurrent callers.

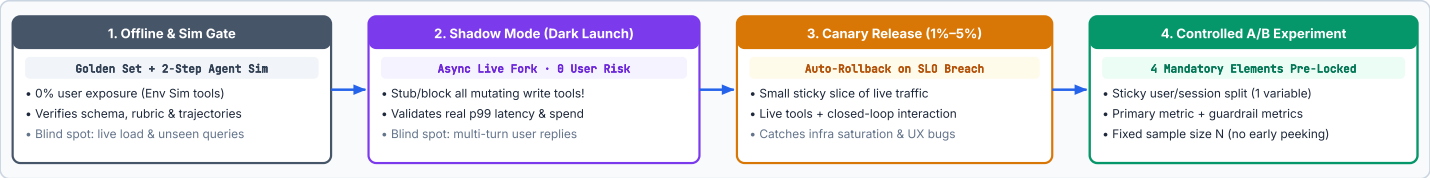
3. **Single Retry Owner:** Official SDKs (`google-genai` , `anthropic`) retry transient errors (`408` , `409` , `429` , `≥500`) **twice by default**. Never wrap an SDK call that has default retries enabled inside your own 4-attempt loop (multiplying to 3 × 4 = 12 hits per request). Either configure `HttpRetryOptions` / `max_retries` on the SDK client or set `max_retries=0` and manage retries in one wrapper.
4. **Never retry terminal errors or content decisions:** Fail fast on `400` (bad schema, invalid `thinking_level`), `401` / `403` (auth), `404` (bad model ID), `413` (payload too large), **workspace spend-cap `429` s** (which lack `retry-after`), and `HTTP 200` with `stop_reason: "refusal"` / safety block.
5. **Idempotency Keys on Mutating Tools:** Retrying a mutating tool (`charge_credit_card` , `issue_refund` , `create_jira_ticket`) after a `504 Gateway Timeout` risks duplicate execution if the first request committed server-side. Every mutating tool must pass a deterministic `idempotency_key` (`sha256(session_id + turn_index + tool_name + canonical_args)`) deduplicated in Memorystore/Redis or the downstream API.

Layers 2–4: Circuit Breakers, Validated Fallbacks & Context-Rich Escalation

- **Layer 2 — Circuit Breakers (`Closed → Open → Half-Open`):** Per-request retries during a 10-minute outage hold thousands of sockets open, starving web workers. A shared service-boundary circuit breaker trips `Open` once error rates breach threshold (failing fast in `<1 ms` into Layer 3), then probes recovery via `Half-Open` after a cooldown window.
- **Layer 3 — Orchestration Fallbacks:** Fail over first to the same model via the `global` **endpoint** or a secondary region; if falling back across model tiers/families (e.g., `gemini-4-argon` → `gemini-3.8-flash` or `Claude Opus` → `Sonnet`), maintain a **fallback-specific prompt/parameter configuration** (adjusted via **Data-Driven Prompt Optimizer** and validated on your Level 2/3 golden suite) so unsupported parameters (like `thinking_budget` vs. `thinking_level`) do not cause secondary `400` crashes.
- **Layer 4 — Structured Human Escalation:** When fallbacks exhaust or `max_turns` / policy boundaries trigger, hand off a structured bundle (`trace_id` , session history, verified customer entities, tools already invoked + outputs, and escalation reason) so the human specialist never asks the customer to repeat themselves.

9. Safe Rollouts: Shadow Mode vs. 4-Element A/B Testing

Passing an offline golden dataset and simulation suite (Levels 1–3) is necessary but never sufficient for high-volume production changes. Real user traffic contains live concurrency, cache warmth, and long-tail phrasing that static suites miss. Promote changes through a **4-Stage Progressive Delivery Pipeline**:



Shadow Mode (Dark Launch) vs. Canary vs. A/B Testing

Rollout Stage	Who Sees Candidate Output?	Side-Effecting Tools Allowed?	What It Validates Best	Blind Spots / Limitations
Offline & Simulation	Eval harness only	Mocked / **Environment Simulation**	Regressions on known golden cases, multi-turn persona stress tests, fault-injection handling	Cannot test real production concurrency, live cache hit ratios, or organic user reactions.
Shadow Mode *(Dark Launch)*	**Nobody** (logged async; user receives baseline response)	**Read-only tools only** (mutating tools MUST be stubbed/sandboxed!)	Real-world `p95` / `p99` latency, crash rates, token spend, cache hit rate, first-turn tool selection, and async autorater parity	**Cannot measure multi-turn user reaction** (user replies to baseline Turn 1, creating counterfactual divergence on Turn 2+) or business conversion.
Canary Release	**1%–5% of live traffic** (sticky by session)	Live tools (with circuit breakers & auto-rollback)	Production stability, quota saturation, and catastrophic UX failures under real closed-loop dialogue	Small slice lacks statistical power to prove subtle (2%–3%) resolution or conversion gains.
Controlled A/B Test	**Randomized split** (e.g., 50/50 or 80/20 sticky by user/account)	Full live stack	Causal impact on primary business/quality outcomes + non-inferiority against cost and latency guardrails	Requires pre-committed sample size N and business-cycle duration; exposes treatment cohort to candidate.

CRITICAL SHADOW MODE SAFETY RULE

In shadow mode, live requests are duplicated asynchronously to the candidate agent. If the candidate calls **mutating tools** (`refund_order`, `update_crm`, `send_email`), running unsandboxed shadow mode executes real side effects twice or corrupts production data! Enforce a **read-only tool sandbox** in shadow mode.

Rigorous Controlled Experimentation & A/B Testing

An A/B experiment must lock in **all four elements before launch**:

1. **Explicit Falsifiable Hypothesis & Sticky Randomization Unit:** Randomize at the **user, account, or session level** (`hash(user_id + experiment_salt) % 100`), **never per HTTP turn** (which mixes models mid-conversation, destroys prefix caching, and ruins causal attribution).
2. **Single-Variable Isolation:** Change **one variable per treatment arm** (e.g., prompt revision OR model tier, not prompt + model + retriever + UI simultaneously).
3. **Primary Outcome Metric + Guardrail Metrics (Non-Inferiority Bounds):** Pair one primary success metric (e.g., *Verified Resolution Rate without Escalation*) with strict guardrails that must not degrade: `p95` TTFT/turn latency, **token cost per resolved task**, unhelpful refusal rate, tool error rate, and `HALLUCINATION` monitor score.
4. **Pre-Committed Sample Size N & Statistical Stopping Rule:** Fix $\alpha = 0.05$, power $1 - \beta = 0.80$, MDE, and minimum full weekly cycle beforehand. **Never stop early the first day p < 0.05 ("peeking" / p-hacking)** unless using formal sequential testing boundaries (Always-Valid p-values / O'Brien-Fleming alpha-spending).

10. Token Economics, Heavy-Tail Sizing & 4-Step Enterprise ROI

More enterprise AI pilots fail at the FinOps gate than at the accuracy gate. A prototype that costs $0.02 per query on clean developer tests can easily cost $0.85 per resolved ticket in production once heavy-tail context growth, ~15× multi-agent fan-out, online judge sampling, and human review labor are included.

Heavy-Tail Token Distributions (`p95` / `p99` vs. `p50`)

LLM token consumption follows a **heavily right-skewed (log-normal / Pareto) distribution**:

- **Why `p50` (median) sizing fails:** While the median session might be 2 turns consuming 3,000 tokens ($0.01), the top 5%–10% (`p95` – `p99`) involve 15-turn debugging loops, 50-page PDFs, large MCP JSON payloads, `thinking_level="HIGH"` reasoning traces, and retries—consuming 150,000 to 500,000+ tokens per session.
- Without context compaction and prefix caching, resending history across N turns bills **quadratically** ($O(N^2)$):

$$\text{Cumulative Input Tokens across N turns} \approx N \cdot S_{static} + \sum_{t=1}^{N} (N - t + 1) \cdot \Delta_{turn\ t}$$

- **FinOps Sizing Rule:** Model TPM quotas and monthly budgets using the **distribution mean or `p95` / `p99`-weighted buckets**, never `p50 × volume`, and enforce hard caps (`max_turns`, `max_tokens`, observation pruning) to clip runaway tails.

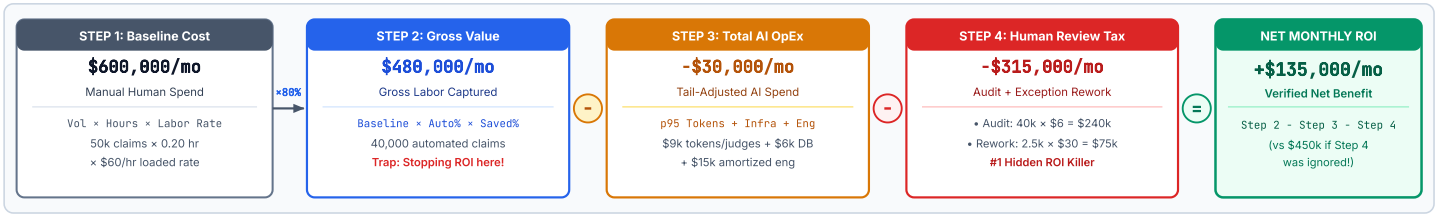
High-Impact Cost-Bending Levers (Multi-Agent ~15× Multiplier)

Autonomous multi-agent systems consume **~15× more tokens** than single-turn chat (~4× more than single-agent tool loops) due to parallel subagent context initialization, multi-hop reasoning, and coordinator synthesis. Bend the cost curve with six architectural levers:

Cost-Optimization Lever	Mechanism	Typical Savings Impact	Critical Exam & Production Nuance
1. Tiered Semantic Model Routing	Route simple classification/guardrails to `gemini-3.5-flash-lite` (`Haiku`), workhorse tool loops to `gemini-3.8-flash` (`Sonnet`), and complex synthesis to `gemini-4-argon` (`Opus`).	**50% – 80%** token spend reduction	Validate each routed tier against your Level 2/3 golden set (and tune prompts with **Data-Driven Optimizer**).
2. Prefix-Aligned Context Caching	**Implicit caching** (automatic on `gemini-4-argon` at ≥ 4,096 tokens and `3.x` at ≥ 4,096 to 6,144 tokens, **zero storage fee**, ~90–95% read discount) vs. **Explicit caching** (`cachedContents` / `cache_control` with TTL storage/write cost where supported).	**Up to 80% – 95%** on repeated input prefixes + lower TTFT	Cache creation bills at standard input price on Gemini; **explicit `cachedContents` pays TTL storage fees while implicit caching has zero storage cost** (`gemini-4-argon` supports implicit only). Keep static instructions/tools at byte 0!
3. Thinking / Effort Right-Sizing	Use `thinking_level="minimal"` on `3.5-flash-lite` or `"low"` / `"medium"` on `3.8-flash` & `4-argon` (and `effort: "low"` on Claude) for routine hops, since thinking tokens bill at **output token rates**.	**30% – 60%** output token reduction	Reserve `thinking_level="high"` for ambiguous planning, root-cause analysis, or complex math/code synthesis.
4. Async Batch / Flex Inference	Run nightly 10,000-case offline evalsets, bulk backfills, and large autorater jobs via **Batch Prediction / Flex PayGo / Message Batches API** (≤ 24h SLA).	**~50% price reduction** on input & output tokens	**Provisioned Throughput (GSUs) does NOT cover batch prediction**—use Batch/Flex for offline workloads and GSUs for steady real-time traffic.
5. Observation Masking & Session Compaction	Prune verbose MCP JSON/HTML fields after extraction (`after_tool_callback`), run tabular filtering inside the **Code Execution sandbox**, and compact older session turns.	**40% – 70%** reduction in multi-turn $O(N^2)$ growth	Store critical IDs/constraints in structured `session.state` before compacting prose to prevent **state erasure**.
6. Idle Infra Hygiene	Tune `min_instances` , suspend idle Cloud Workstations (`idleAction=SUSPEND`), and undeploy unused **Vector Search 1.0** index endpoints.	Eliminates 24/7 fixed hourly burn at zero QPS	Deployed Vector Search 1.0 endpoints bill continuously per node-hour even when serving zero queries.

Unit Economics & Total Cost of Ownership (TCO) Ledger

Calculate **Net Monthly Benefit** across all four steps—ignoring **Step 4 (Human-Review & Exception Tax)** is the #1 cause of negative-ROI production launches:



- **Worked Sanity Check:** On 50,000 claims/month at $12/claim baseline ($600k), 80% automation yields **$480k Gross Savings** (Step 2) against **$30k AI OpEx** (Step 3). If compliance requires a 6-minute ($6) audit on all 40,000 automated claims ($240k) plus 20-minute ($30) specialist rework on 2,500 subtle errors ($75k), **Step 4 Review Tax = $315k**, cutting Net Benefit from $450k down to **$135k/month**. If audit time approaches baseline manual time (e.g., 14 min vs. 12 min), Net ROI turns **negative**—fixable only by raising `pass^k` precision, adding $0 Tier 1 deterministic gates, and calibrating confidence routing so humans audit only genuinely uncertain slices.

Architectural Readiness & Launch-Gate Triage

Launch-Gate Verdict	Engineering & Economic Criteria	Canonical Scenario	Architectural Action
1. Production-Ready (Go)	Accuracy, `p95` latency, unit economics, and governance bounds all fit current models and data.	Internal IT helpdesk agent grounded on Agent Search with 4s streaming TTFT and ticket fallback.	Proceed to Level 1–3 eval harness + staged canary rollout.
2. Conditional Go (Guardrails Required)	Core task works, but synchronous latency or unreviewed autonomy conflicts with latency budgets or risk posture.	50-page contract redlining requested in `< 1s` with 100% autonomous legal sign-off.	Re-architect around **async background execution** (`background=True` , 30–60s) + **mandatory human approval gate (HITL)**.
3. No-Go (Use Deterministic Automation)	Requires 100.00% zero-defect determinism without review, `< 5 ms` control loops, or 10× negative unit economics.	High-frequency `< 5 ms` trading execution or unreviewed autonomous clinical dosing from raw handwriting.	**Keep LLMs off the critical path**; use deterministic rules/classical ML, or limit AI strictly to assistive drafting with HITL.

11. Production Failure Modes & Diagnostic Matrix

Production Failure Mode Diagnostic Matrix

Observed Production Symptom	Root Cause	Correct Engineering Remediation	Trap / Wrong Fix to Avoid
Compliance agent scores 97% on `FINAL_RESPONSE_QUALITY` yet issues refunds without checking eligibility	**"Right answer, wrong path"** blind spot in outcome-only grading	Pair outcome autoraters with **`trajectory_in_order_match`** / **`trajectory_exact_match`** (`trajectory_single_tool_use`) and a forbidden-tool `CustomMetric`	Raising the `FINAL_RESPONSE_QUALITY` threshold from 0.90 to 0.99
Online Evaluation Monitor rejects `trajectory_exact_match` or causes a 6×–31× judge LLM billing spike on live traffic	Live traffic has **no ground-truth reference trajectory**, and 100% sampling triggers **5–31 judge calls/trace** (`TOOL_USE_QUALITY` ~5× Flash + 1× Pro; `SAFETY` `safety_v3` **31× Flash**)	Score **reference-free rubrics** (`HALLUCINATION` , `TOOL_USE_QUALITY` , `FINAL_RESPONSE_QUALITY`) with **trace sampling filters** (`5%` sample or `duration > 5s`) + max samples/run	Trying to attach `trajectory_exact_match` online or enabling 100% online rubric evaluation on high-QPS traffic
Cloud Monitoring hits custom metric cardinality limits and spikes observability spend after tracking per-session tokens	Unbounded **`session_id`** (**`gen_ai.conversation.id`**) or **`user_id`** attached as a Cloud Monitoring metric label	Strip high-cardinality IDs from Monitoring labels (group only by bounded `gen_ai.tool.name` , `model` , `status`); compute `p99` tokens/session in **Log Analytics / BigQuery log sink**	Hashing `session_id` into another unbounded Cloud Monitoring label
On-call engineers get paged on 30-second `503` blips while a slow `0.3%` error leak silently burns the monthly 99.9% SLO	Alerting on single-window static error counts or arithmetic mean latency	Use **multi-window SLO burn-rate alerts** (`select_slo_burn_rate(..., "3600s") > 10.0` fast burn + `"86400s" > 2.0` slow burn) and **`ALIGN_PERCENTILE_95`** latency aligners	Lowering a single 5-minute static error-rate threshold until alert fatigue worsens
Team has zero historical chat logs and cannot test multi-turn edge cases or `503` tool outages safely	Relying solely on static single-turn prompts or live prod backends	Run **2-Step Agent Simulation** (from an **Agent Registry** snapshot) + **Environment Simulation** fault injection	Running load/fault tests directly against live production payment APIs
Manual prompt edits to fix one bug cluster keep breaking three unrelated instructions	Manual "prompt whack-a-mole" without closed-loop optimization	Cluster traces via **Failure Cluster Analysis**, expand the golden set (with a **held-out split**), and run **Data-Driven Prompt Optimizer / Agent Optimizer**	Appending contradictory ALL-CAPS warnings to the bottom of the system prompt
Deploying a custom Streamable HTTP MCP server onto Agent Runtime fails	**Agent Runtime** hosts stateful agent orchestrators (`reasoningEngines`), not standalone MCP servers	Host custom Streamable HTTP MCP servers on **Cloud Run** (or **GKE**) and register them in **Agent Registry**	Embedding external MCP server daemons inside `session.state`
Users lose conversation context randomly on a TypeScript/Python agent hosted on Cloud Run	Storing session history in **ephemeral Cloud Run container RAM** across scale-to-zero recycles	Externalize state to **Agent Platform Sessions** (`agentengine://`), **Memorystore for Redis**, **Firestore**, or **Cloud SQL**	Increasing Cloud Run request timeout to 60 minutes without external state
Architect proposes self-hosting `gemini-4-argon` on GKE GPUs for VPC isolation	**Proprietary Gemini weights cannot be self-hosted** on customer clusters	Call managed Gemini endpoints inside **VPC Service Controls** (+ Provisioned Throughput if needed), or self-host open-weight **Gemma 4** on GKE	Creating a GKE GPU node pool to pull `gemini-4-argon` container weights
Agent Runtime deployment fails with `"Region temporarily at capacity"`	**Physical regional compute pool exhaustion** (not a project quota limit)	Deploy to an alternate supported region, reduce per-instance vCPU/RAM, or retry off-peak	Filing an IAM or project quota-increase support ticket and waiting
Every component passes isolated Level 2 evals at >95%, but end-to-end tasks fail 45% of the time	**Integration seam failure** (schema mismatch, oversized tool payloads, or state loss during compaction)	Add **Level 3 Trajectory & Seam Integration Tests**, strict JSON schemas, and structured `session.state` preservation	Tuning isolated Level 2 prompts further in single-step harnesses
Shadow mode rollout accidentally sends duplicate cancellation emails to real customers	Shadow pipeline executed **live mutating tools** instead of a read-only sandbox	Intercept/mock all state-mutating tools in shadow mode; allow only read-only retrieval	Running unsandboxed shadow mode on a smaller 5% traffic slice
AI saves 80% of drafting time, yet total department labor cost rises by 12%	**Step 4 Human-Review & Exception Tax** exceeds Step 2 Gross Savings	Add $0 Tier 1 code validators and calibrate confidence routing so humans audit only uncertain/high-risk slices	Ignoring human audit time in the FinOps ROI spreadsheet

12. Glossary

Term	Definition
3-Tier Grading Hierarchy	Evaluation ladder ordering checks from **Tier 1** (deterministic code/schema assertions, $0) → **Tier 2** (calibrated LLM autoraters) → **Tier 3** (human domain experts).
6 Deterministic Trajectory Metrics	`$0` -judge-cost offline Gen AI Evaluation metrics requiring a ground-truth reference trace: `trajectory_exact_match` , `trajectory_in_order_match` , `trajectory_any_order_match` , `trajectory_precision` , `trajectory_recall` , and `trajectory_single_tool_use` .
Adaptive Rubrics	Reference-free Gen AI Evaluation Service metrics (`FINAL_RESPONSE_QUALITY` , `TOOL_USE_QUALITY` ~5× Flash + 1× Pro, `MULTI_TURN_TASK_SUCCESS` , `MULTI_TURN_TOOL_USE_QUALITY` , `MULTI_TURN_TRAJECTORY_QUALITY` , `INSTRUCTION_FOLLOWING` , `GENERAL_QUALITY`) that synthesize prompt-specific pass/fail unit tests per input.
`CustomMetric` vs. `types.LLMMetric`	Custom Gen AI Evaluation primitives: `CustomMetric` runs deterministic Python checks (regex, SQL AST, schema) at `$0` judge cost; `types.LLMMetric` scores outputs with a custom LLM judge rubric (`PointwiseMetric` / `PairwiseMetric`).
2-Step Agent Simulation	Multi-turn synthetic stress-testing from an **Agent Registry** snapshot: Step 1 synthesizes `Starting Prompts` + `Conversation Plans` ; Step 2 executes simulated user dialogues.
Environment Simulation	Evaluation harness layer that intercepts outbound tool calls to inject deterministic faults (`HTTP 503` , `429` , timeouts, mocked rows) without touching live backends.
Online Evaluation Monitors	Continuous production evaluators (`OnlineEvaluator`) that asynchronously sample live Cloud Trace spans (`5%` or `duration > 5s` filter) and score them with reference-free autoraters (`HALLUCINATION` , `TOOL_USE_QUALITY` , `SAFETY`).

Term	Definition
`select_slo_burn_rate`	Cloud Monitoring filter function measuring SLO error-budget consumption speed over a lookback window; paired as a `3600s` `10×` fast-burn page and an `86400s` `2×` slow-burn ticket.
Metric Cardinality Explosion	Observability anti-pattern caused by attaching unbounded `session_id` (`gen_ai.conversation.id`), `user_id` , or `trace_id` values as Cloud Monitoring metric labels instead of querying them in Log Analytics / BigQuery.
Failure Cluster Analysis	Automated LLM grouping of failed simulation and low-scoring production traces into ranked root-cause buckets (Frequency × Severity).
Automatic Loss Analysis	Agent Platform feature that reads pass/fail signals from rubric-based metrics, classifies failures into predefined loss patterns and groups them into semantic clusters.
Quality alert	Cloud Monitoring alerting policy on online monitor scores (`aiplatform.googleapis.com/online_evaluator/scores`) that notifies owners through Slack, email or Pub/Sub when quality drifts below a threshold.
Agent Anomaly Detection	Agent Platform service (Preview, allowlisted) that analyzes ADK traces and logs asynchronously and reports anomalous sessions to Security Command Center, including infinite loops and oscillating retries under the OWASP ASI08 cascading failures category.
Loop exit signal (`escalate`)	ADK event action (`tool_context.actions.escalate = True` or `EventActions(escalate=True)`) that ends a `LoopAgent` before `max_iterations` is reached.
`DEFAULT_ROUTE`	ADK graph workflow route that matches when no other route from the same source node matches; point it at a terminal or fallback node, not at a loop back-edge.
Topology view	Agent Registry and project-level graph (App Topology API) of inbound and outbound dependencies between agents and MCP servers, with latency and error rates from the most recent hour of trace data.
`--otel_to_cloud`	ADK CLI flag (`adk web` , `adk deploy agent_engine`) that exports the built-in OpenTelemetry instrumentation to Google Cloud Observability.
Message content capture	`OTEL_INSTRUMENTATION_GENAI_CAPTURE_MESSAGE_CONTENT=EVENT_ONLY` records prompts and responses as events (required by online monitors), while `ADK_CAPTURE_MESSAGE_CONTENT_IN_SPANS=false` keeps that text out of span attributes.

Term	Definition
Zero-Shot vs. Data-Driven Optimizer	Agent Platform prompt optimization engines: **Zero-Shot** rewrites a raw prompt immediately using structural guidelines (no data); **Data-Driven / Agent Optimizer** iteratively optimizes `instruction` , `demonstration` , or `instruction_and_demo` against labeled data and eval metrics for a target model.
Dual-Attribution Audit Log	Cloud Logging / Audit Log record (`is_agentic=true`) capturing both the cryptographic SPIFFE **Agent Identity** (`principal://...`) and the initiating human user principal.
Agent Runtime vs. Cloud Run vs. GKE	Google Cloud runtime triad: **Agent Runtime** for turnkey stateful Python/ADK/A2A agents; **Cloud Run** for scale-to-zero polyglot containers and Streamable HTTP MCP servers; **GKE** (`Agent Substrate` / `Agent Sandbox`) for self-hosted **Gemma 4** on GPUs/TPUs and gVisor kernel isolation.
`Evaluation Dataset` (`agents-cli` & Eval SDK)	Gen AI Evaluation SDK dataset type read by `client.evals.evaluate()` (tabular rows / OTel traces) and upgraded `agents-cli eval` scaffolds (`tests/eval/datasets/*-dataset.json` + `eval_config.yaml` , migrated via `agents-cli scaffold upgrade` from earlier `tests/eval/evalsets/*.evalset.json` layouts).
`eval_config.yaml`	`agents-cli` file under `tests/eval/` that lists `metrics_to_run` and defines custom code metrics (`custom_function`) or LLM-as-judge metrics (`prompt_template`).
Evalset	ADK `*.evalset.json` file holding one or more sessions (or a conversation scenario for user simulation), graded by `adk eval` or `AgentEvaluator` with criteria from `test_config.json` .
Golden set	Frozen, versioned dataset of trajectory, multi-turn, abstention, adversarial and backfilled cases that every CI/CD gate grades against.
Agent Runtime contract	Requirements for custom containers on Agent Runtime: listen on `0.0.0.0:8080` , and implement `/api/reasoning_engine` and `/api/stream_reasoning_engine` to support the Python SDK and console playground.
`container_concurrency`	Agent Runtime setting for concurrent requests per instance; defaults to 9 (one per agent process), and async agents usually start at a multiple of 9.
`pass@k` vs. `pass^k`	At-least-one success $(1 - (1 - p)^k$, valid only with a deterministic verifier) vs. every-run consistency (p^k, mandatory for live customer agents and unattended workflows).
Tool Payload Bloat & State Loss	Multi-turn failure modes where huge unpruned tool payloads dilute model attention or aggressive prose summarization drops a critical earlier UUID or parameter.
Human-Review Tax (Step 4 ROI)	Mandatory labor cost of human auditors reviewing flagged AI outputs and specialists fixing false positives—subtracted in Step 4 of enterprise ROI.